UNA GUIDA DEDICATA A TUTTI GLI SVILUPPATORI C++ DESIDEROSI DI SCRIVERE APPLICAZIONI PER IL FRAMEWORK MICROSOFT .NET

APPLICAZIONI .NET CON C++/CLI

Antonio Pelleriti

APPLICAZIONI .NET CON C++/CLI

di Antonio Pelleriti

INDICE

INTRODUZIONE

Questo libro è una guida alla programmazione in C++/CLI, il nuovo linguaggio che consentirà ai numerosi programmatori C++ sparsi nel mondo di sviluppare applicazioni per .NET, da 2.0 in su, senza avere nulla da invidiare agli sviluppatori C#, VB.NET, o a quelli che utilizzano uno dei tanti linguaggi ormai supportati dal framework di programmazione di casa Microsoft. Anzi, C++/CLI si candida, ed ha tutte le carte in regola, per diventare uno degli attori protagonisti sul palcoscenico dello sviluppo software .NET.
C++ è stato ed è tuttora uno dei linguaggi più utilizzati e più potenti. Con C++ si può realizzare qualunque tipo di applicazione, e forse l'unica mancanza, a meno di un tentativo poco elegante a dir poco come le managed extensions, era l'impossibilità di sviluppare applicazioni .NET.
Prima di rispondere alla domanda che molti di voi si sono probabilmente posti, quando hanno visto per la prima volta il termine C++ accostato alla sigla CLI, è meglio partire da cosa siano separatamente C++ e CLI. C++/CLI rappresenta una tupla. Il primo termine è C++, che si riferisce al linguaggio C++ standard inventato da Bjarne Stroustrup. C++ supporta un modello ad oggetti statico ottimizzato per la velocità di esecuzione e le prestazioni. Consente poi l'accesso al sistema operativo e all'hardware sottostante, ma dà solo un piccolo supporto per accedere ai tipi attivi durante l'esecuzione. CLI si riferisce invece alla Common Language Infrastructure, l'architettura multi strato che è un componente fondamentale di .NET e che supporta il modello di programmazione dinamico ed a componenti. Lo slash (/) che unisce rappresenta proprio il collegamento dra C++ e CLI.
Nel libro ci si riferirà al nuovo linguaggio sempre con C++/CLI, mentre quando si farà riferimento a C++ si intenderà il "vecchio" C++. Naturalmente, una guida introduttiva di queste dimensioni non può esplorare a fondo un linguaggio di programmazione complesso e

potente come C++/CLI, dunque spero che questo testo serva a solleticare l'appetito del lettore e magari portarlo ad approfondire l'argomento. Questo è il mio terzo libro, e come i primi due lo dedico a Caterina, ringraziandola per la sua pazienza e per quelle volte che, invece di uscire per una pizza, sono rimasto a scrivere classi e handle. Parlando di C++/CLI cambio sintassi: while(1){me->TiAdoro();};

IL FRAMEWORK .NET

L'architettura del framework .NET è a strati sovrapposti, e il layer inferiore poggia direttamente al sistema operativo, non necessariamente Microsoft Windows, dato che esistono e sono anche a buon punto progetti per lo sviluppo in .NET su ambienti diversi, ad esempio Mono. Il framework .NET consiste di cinque componenti fondamentali: il primo è il Common Language Runtime (CLR). Esso fornisce le funzionalità fondamentali per l'esecuzione di un'applicazione managed. Il CLR, a basso livello, si occupa inoltre dell'interfacciamento con il sistema operativo. Lo strato immediatamente al di sopra del CLR è costituito dalla Base Class Library (o .NET Framework Class Library) di .NET, cioè un insieme di classi fondamentali, utili a tutte le applicazioni ed a tutti gli sviluppatori. La BCL contiene i tipi primitivi, le classi per l'Input/Output, per il trattamento delle stringhe, per la connettività, o ancora per creare collezioni di oggetti. Dunque, per chi avesse esperienza con altre piattaforme, può essere pensato come un insieme di classi analogo a MFC, VCL, Java. Naturalmente altre classi specializzate saranno sicuramente mancanti nella BCL. Al di sopra della BCL, vengono quindi fornite le classi per l'accesso alle basi di dati e per la manipolazione dei dati XML che, come vedrete iniziando a fare qualche esperimento, sono semplici da utilizzare ma estremamente potenti e produttive. Lo strato più alto del framework è costituito da quelle funzionalità che offrono un'interfacciamento con l'utente finale, ad esempio classi per la creazione di interfacce grafiche desktop, per applicazioni web, o per i sempre più diffusi web service.

CLR

Il componente più importante del framework .NET è, come già

detto, il Common Language Runtime, che gestisce l'esecuzione dei programmi scritti per la piattaforma .NET. Chi ha avuto modo di sviluppare in Java può paragonare il CLR alla Java Virtual Machine che esegue bytecode Java.
Il CLR si occupa dell'istanziazione degli oggetti, esegue dei controlli di sicurezza, ne segue tutto il ciclo di vita, ed al termine di esso esegue anche operazioni di pulizia e liberazione delle risorse.
In .NET ogni programma scritto in un linguaggio supportato dal framework viene tradotto in un linguaggio intermedio comune, detto CIL (Common Intermediate Language) o brevemente IL, ed a questo punto esso può essere tradotto ed assemblato in un eseguibile .NET, specifico per la piattaforma su cui dovrà essere eseguito. In effetti, a run-time, il CLR non conosce e non vuole conoscere in quale linguaggio lo sviluppatore ha scritto il codice, il risultato della compilazione, è un modulo managed, indipendente dal linguaggio utilizzato, addirittura è possibile scrivere le applicazioni direttamente in linguaggio IL.
Un modulo managed contiene sia il codice IL che dei metadati. I metadati non sono altro che delle tabelle che descrivono i tipi ed i loro membri definiti nel codice sorgente, oltre ai tipi e relativi membri esterni referenziati nel codice.
Il CLR però non esegue direttamente dei moduli, ma lavora con delle entità che sono chiamate assembly. Un assembly è un raggruppamento logico di moduli managed, e di altri file di risorse, ad esempio delle immagini utilizzate nell'applicazione, dei file html o altro ancora, ed in aggiunta a questi file, ogni assembly possiede un manifesto che descrive tutto il suo contenuto, ciò rende possibile il fatto che ogni assembly sia una unità autodescrittiva.
I compilatori .NET, ad esempio il compilatore C#, quello VB.NET, o quello C++/CLI, attualmente creano un assembly in maniera automatica a partire dai moduli, aggiungendo il file manifesto.
Un assembly può essere non solo un'eseguibile, ma anche una libreria DLL contenente una collezione di tipi utilizzabili eventua-

mente in altre applicazioni.

LA COMPILAZIONE JIT

Il codice IL non è eseguibile direttamente dal microprocessore, e nemmeno il CLR può farlo, in quanto esso non ha funzioni di interprete. Dunque esso deve essere tradotto in codice nativo in base alle informazioni contenute nei metadati. Questo compito viene svolto a tempo di esecuzione dal compilatore JIT (Just In Time), altrimenti detto JITter.
Il codice nativo viene prodotto quando necessario. Ad esempio la prima volta che viene invocato un metodo, esso viene compilato, e conservato in memoria. Alle successive chiamate il codice nativo sarà già disponibile in cache, risparmiando anche il tempo della compilazione just in time.
Il vantaggio della compilazione JIT è che essa può essere realizzata dinamicamente in modo da trarre vantaggio dalla caratteristica del sistema sottostante. Ad esempio lo stesso codice IL può essere compilato in una macchina con un solo processore, ma potrà essere compilato in maniera differente su una macchina biprocessore, sfruttando interamente la presenza dei due processori. Ciò implica anche il fatto che lo stesso codice IL potrà essere utilizzato su sistemi operativi diversi, a patto che esista un CLR per tali sistemi operativi.

COMMON TYPE SYSTEM

Dati i differenti linguaggi che è possibile utilizzare per scrivere codice .NET compatibile, è necessario avere una serie di regole atte a garantirne l'interoperabilità, la compatibilità e l'integrazione dei linguaggi. Una classe scritta in C++/CLI deve essere utilizzabile in ogni altro linguaggio .NET, ed il concetto stesso di classe deve essere uguale nei diversi linguaggi, cioè una classe come

intesa da C#, deve essere equivalente al concetto che ne ha VB.NET oppure C++/CLI, o un altro linguaggio .NET enabled. Per permettere tutto questo, Microsoft ha sviluppato un insieme di tipi comuni, detto Common Type System (CTS), suddivisi in particolare, in due grandi categorie, tipi riferimento e tipi valore ma, come vedremo più in là nel testo, ogni tipo ha come primo antenato un tipo fondamentale, il tipo Object.
Il Common Type System definisce come i tipi vengono creati, dichiarati, utilizzati e gestiti direttamente dal CLR, e dunque in maniera ancora indipendente dal linguaggio. D'altra parte, ogni linguaggio ha caratteristiche distintive, in più o in meno rispetto ad un altro. Per garantire l'integrazione fra i linguaggi è necessario stabilire delle regole, e nel far ciò Microsoft ha creato una specifica a cui tali linguaggi devono sottostare. Tale specifica è chiamata Common Language Specification (CLS). Naturalmente ogni linguaggio può anche utilizzare sue caratteristiche peculiari, e che non sono presenti in altri, in questo caso però il codice non sarà accessibile da un linguaggio che non possiede quella particolare caratteristica, nel caso contrario, cioè nel caso in cui, ad esempio, un componente è scritto facendo uso solo di caratteristiche definite dal CLS,. allora il componente stesso sarà detto CLS-compliant. Lo stesso CTS contiene tipi che non sono CLS-compliant, ad esempio il tipo UInt32, che definisce un intero senza segno a 32 bit, non è CLS compliant, in quanto non tutti i linguaggi hanno il concetto di intero senza segno.

GESTIONE DELLA MEMORIA

In linguaggi come C e C++, lo sviluppatore si deve occupare in prima persona della gestione della memoria, cioè della sua allocazione prima di poter creare ed utilizzare un oggetto e della sua deallocazione una volta che si è certi di non dover più utilizzarlo. Il CLR si occupa della gestione della memoria in maniera au-

tomatica, per mezzo del meccanismo di garbage collection. Tale meccanismo si occupa di tener traccia dei riferimenti ad ogni oggetto creato e che si trova in memoria, e quando l'oggetto non è più referenziato, cioè il suo ciclo di vita è terminato, il CLR si occupa di ripulirne le zone di memoria a questo punto non più utilizzate.
Tutto ciò libera il programmatore da buona parte delle proprie responsabilità di liberare memoria non più utilizzata, e d'altra parte dalla possibilità di effettuare operazioni pericolose nella stessa memoria, andando per esempio a danneggiare dati importanti per altri parti del codice.

MANAGED O NATIVO?

Il cosiddetto managed code è il codice compilato appositamente per essere eseguito all'interno del Common Language Runtime di .NET e quindi che sfrutta la sua gestione della memoria, la code access security, l'integrazione multi linguaggio. Al contrario, il native code, conosciuto anche come unmanaged code, non ha come target il CLR, ma gira al di fuori di esso, direttamente all'interno del sistema operativo. A partire da .NET 2.0 è possibile creare del safe code, che significa codice managed verificabile dal CLR. Viceversa, unsafe code è codice che non può essere verificato, e quindi generalmente codice scritto in formato nativo.
Il .NET Framework fornisce dei compilatori, per esempio quelli per C# e VB.NET, che generano codice managed. Il compilatore C++/CLI invece per default non lo fa, e deve essere impostato da riga di comando oppure utilizzando le opzioni di Visual Studio.
Per creare codice managed è possibile utilizzare queste opzioni:

- */clr:oldSyntax*: compila il codice C++/CLI scritto con sintassi .NET versione 1.x e crea generalmente un'immagine mista di codice managed e nativo.
- */clr*: questa opzione è utilizzata per la nuova sintassi C++/CLI di .NET 2.0. Anche in questo caso si crea una immagine mista di codice nativo e gestito.
- */clr*: pure: l'opzione è usata per generare codice managed e dati unmanaged. Utilizzando codice unsafe all'interno di un programma la compilazione fallisce.
- */clr*: safe: l'opzione è utilizzata per generare un programma interamente managed. Utilizzando codice unsafe e/o dati unmanaged la compilazione fallisce.

Utilizzando Visual Studio per creare programmi, e selezionando uno dei template appositamente aggiunti per i progetti C++/CLI, verrà imposta-

ta automaticamente l'opzione */clr*, ma è possibile impostare ad esempio l'opzione */clr:safe* se si vuole creare esclusivamente codice managed.
Per farlo basta cliccare con il tasto destro sul nome del progetto, andare nella scheda General, e poi alla riga *Common Language Runtime Support* selezionare una delle quattro opzioni elencate sopra.

SINTASSI E SEMANTICA DI C++/CLI

Prima di potere essere in grado di creare programmi utili e destinati al mondo reale, è necessario imparare a conoscere gli elementi che costituiscono un qualsiasi linguaggio. Nei prossimi paragrafi si partirà dal classico programma Hello World, e mano a mano si vedrà cosa sono variabili, tipi di dati, istruzioni, espressioni, costrutti per creare cicli, per controllare il flusso di un programma e così via, coprendo tutto ciò che costituisce la sintasi e la semantica di C++/CLI.

HELLO C++/CLI

Come ogni libro di programmazione che si rispetti, si inizierà mostrando il classico programma che stampa la stringa Hello World sulla console.
Se non si ha a disposizione Visual Studio, anche in edizione Express, è per ora sufficiente un editor di testo qualsiasi.
Basta ricopiare il seguente codice e salvarlo con un nome tipo *helloworld.cpp*:

```
// HelloWorld.cpp : il programma Hello World
using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    return 0;
}
```

Per compilare il programma, si apra un prompt dei comandi di Visual Studio, oppure si aggiunga al path il percorso del compilatore *cl.exe*.
Per creare il file eseguibile *helloworld.exe* basta lanciare il comando seguente:

```
cl Hello.cpp /clr:safe /doc
```

L'eseguibile che si ottiene, se non si sono commessi errori, è un eseguibile .NET che esegue all'interno del Common Language Runtime. Con il termine eseguibile .NET si intende naturalmente un programma MSIL che viene compilato Just-In-Time ed eseguito dal CLR proprio come i programmi scritti in altri linguaggi supportati da .NET, come C# o VB.NET. Con ciò è quindi chiaro che utilizzando una sintassi familiare ai programmatori C e C++ è possibile scrivere applicazioni .NET, ed inoltre sarà possibile utilizzare anche codice nativo nella stessa applicazione.
Prima di cominciare ad analizzare i vari costrutti, si darà una spiegazione rapida del programma appena creato, in maniera da cominciare ad avere un'idea, per chi è alle prime armi, di come è fatto un qualunque programma C++/CLI.
La prima istruzione è un commento, non fa niente durante l'esecuzione, serve solo a inserire istruzioni o del testo per mostrare come funziona un programma.
L'istruzione *using* indica al programma di utilizzare le funzionalità contenute nello spazio dei nomi System. Si vedrà più avanti in maniera più approfondita cos'è un namespace, per ora basta pensarlo come un contenitore di classi che possiamo utilizzare nel codice. In questo caso si ha bisogno di scrivere sulla console una stringa, cioè una sequenza di caratteri. Il namespace *System* contiene la classe *Console*, che verrà utilizzata all'interno del metodo principale del programma. Il metodo main è il punto di ingresso del programma, come spiegato nel successivo paragrafo.
Il corpo di un metodo è un blocco di istruzioni delimitato dalle parentesi graffe, all'interno del blocco viene utilizzato il metodo *WriteLine* della

classe Console, passandogli come argomento la stringa che si vuol stampare, in questo caso "Hello World".Subito dopo, come ultima istruzione del programma, viene restituito il valore 0.

IL METODO MAIN

Ogni programma C++/CLI deve avere un metodo *main()*, che rappresenta il punto di ingresso del programma stesso, e tale metodo deve quindi essere uno ed uno solo. Ci sono diverse possibilità con cui scrivere un metodo *main*. Nell'esempio *hello world* del paragrafo precedente il metodo *main* restituisce un valore intero, e prende come argomento un array di stringhe. Questo è il modo più completo di scrivere il metodo *main*:

```
int main(array<System::String ^> ^args)
{
    return 0;
}
```

Se non è necessario passare degli argomenti al programma dalla riga di comando, è possibile evitare l'utilizzo dell'array args, e analogamente se non si vuol restituire un valore all'esterno, si può usare come tipo *void* anziché *int*:

```
void main()
{}
```

Quello precedente è un perfetto esempio di programma inutile, non gli si può passare alcun argomento, non fa niente e non restituisce niente all'esterno.

ARGOMENTI DEL MAIN

Come detto qualche riga fa, è possibile passare degli argomenti ad un pro-

gramma dalla linea di comando. Tali argomenti, vengono passati al metodo main in un vettore di stringhe, eventualmente vuoto. Non abbiamo ancora trattato gli array in C++/CLI, ma per il momento basterà dare un'occhiata ad un semplice esempio per capire come utilizzarli in questo caso:

```
using namespace System;

int main(array<System::String ^> ^args)
{
      if(args->Length>0)
      {
            Console::WriteLine("Hello "+args[0]);
      }
   else Console::WriteLine("Hello World");
   return 0;
}
```

Dopo aver compilato il codice, ed aver ottenuto ad esempio il file *hello.exe*, eseguiamolo passando un argomento sulla linea di comando:

```
hello Antonio
```

L'array *args* in questo caso conterrà la stringa *Antonio*, quindi la proprietà *Length* restituirà per l'array una lunghezza pari a *1*. Il metodo *WriteLine* stavolta stamperà la stringa *Hello* seguita dall'elemento 0 dell'array *args*, cioè dal suo primo e unico elemento. È bene sottolineare che gli elementi degli array vengono numerati a partire da zero.

ORGANIZZARE LE CLASSI

Per fornire una organizzazione gerarchica del codice, simile al modo in cui sono organizzate delle directory nel file system, .NET e C++/CLI utilizza-

no il concetto di *namespace*.
A differenza del concetto di directory e file però, il concetto di namespace non è fisico ma solamente logico, non è necessaria cioè corrispondenza fra il nome dei namespace e il nome delle directory in cui sono contenuti.
Il namespace permette di specificare in modo completo il nome di una classe in esso contenuta, separando i nomi dei namespace con i doppi due punti (::) e finendo con il nome della classe stessa.
Per mezzo dei namespace ad esempio possiamo raggruppare classi che si riferiscono o che forniscono funzionalità correlate.

I namespace

Per creare un namespace basta utilizzare la parola chiave *namespace*, seguita da un blocco *{}* all'interno del quale inserire le nostre classi o altri namespace. Ad esempio:

```
namespace Edmaster
{
    Namespace Ioprogrammo
    {
        class HelloWorld
        {
            …
        }
    }
}
```

In questo esempio si sono definiti due namespace annidati, che portano la nostra classe *HelloWorld* ad avere il seguente fullname:

```
Edmaster::Ioprogrammo::HelloWorld
```

Per utilizzare un namespace, evitando così di dover digitare per intero il

nome di una classe o di una funzione, è necessaria un'altra parola chiave di C++/CLI, *using*.
Ad esempio si è già visto che la classe *Console* è contenuta nel namespace *System*, e senza l'utilizzo dell'istruzione:

```
using namespace System;
```

avremmo dovuto scrivere

```
System::Console::WriteLine("Ciao Mondo!");
```

e così per ogni classe contenuta nel namespace *System*. L'utilizzo del solo nome della classe si rende quasi necessario (e senza dubbio comodo) quando abbiamo a che fare con namespace molto annidati e quindi con nomi completi delle classi molto lunghi.

ISTRUZIONI

Ogni programma C++/CLI è composto da una sequenza di istruzioni, che dunque è l'elemento base da combinare per eseguire azioni complesse.
Esistono istruzioni più o meno lunghe e complicate, ma ogniìuna di esse deve essere terminata da un punto e virgola.
Nell'esempio dell'*Hello World* l'istruzione che restituisce il valore 0 è per esempio scritta così:

```
return 0;
```

return è una parola chiave del programma, e 0 è un operando dell'istruzione return. L'istruzione completa viene terminata appunto con un punto e virgola
È possibile combinare più istruzioni in un singolo blocco, raggruppandole all'interno di una parentesi graffa aperta { e una chiusa }.

DICHIARAZIONI E VARIABILI

In C++/CLI, come in quasi ogni linguaggio di programmazione, le dichiarazioni consentono la definizione degli elementi che costituiranno il programma. Si è già visto come avviene la dichiarazione di un namespace, e si è detto che all'interno dei namespace trovano posto le dichiarazioni dei tipi di quel dato namespace. Questi tipi possono essere ad esempio classi, strutture, enumerazioni e così via. All'interno dei tipi, poi, verranno dichiarati altri elementi, come campi e metodi di una classe. Le dichiarazioni sono necessarie, in quanto servono ad assegnare ad ogni elemento del programma un nome univoco, perlomeno nel proprio campo d'azione o *scope*. Ad esempio, scrivendo l'istruzione

```
MioTipo numero;
```

stiamo dichiarando che *numero* è un identificatore di un elemento di tipo *UnTipo*, abbiamo cioè dato un nome ad un elemento del programma.

VARIABILI

In C++/CLI è necessario dichiarare e inizializzare ogni variabile. Una variabile è un identificatore di una zona di memoria, e ogni variabile è associata ad un tipo che identifica quello che la data porzione di memoria dovrà contenere. Nel paragrafo precedente abbiamo già utilizzato una variabile, mediante la forma di dichiarazione essenziale, costituita dal nome del tipo e dal nome della variabile, o identificatore:

```
MioTipo numero;
```

Così facendo si è dichiarata una variabile di tipo *MioTipo*, cioè è stata prenotata e delimitata una zona di memoria adeguata a contenere un *MioTipo*.

Una variabile può essere dichiarata praticamente in qualunque punto del programma, l'unica regola da rispettare è che prima di essere usata, la va-

riabile deve essere dichiarata.
Diamo un'occhiata a degli esempi più reali, anche se ancora non abbiamo parlato dei tipi di C++/CLI.

```
int i;
float f;
```

Le tre istruzioni precedenti dichiarano due variabili *i* ed *f*, di tipo rispettivamente *int* e *float*. Queste dichiarazioni indicano inoltre al compilatore di allocare uno spazio di memoria adeguato a contenere due dati dei relativi tipi. Per mezzo delle variabili potremo quindi accedere alle aree di memoria in cui sono contenuti i dati veri e propri.
C++/CLI permette poi la dichirazione di altri due tipi particolari di dati: i puntatori, croce e delizia dei programmatori C/C++, e gli handle, novità introdotta per .NET.
Si vedrà più in là il significato di tali termini e tali tipi, per il momento si mostrerà come dichiararli, in maniera da poterli riconoscere fin da ora. Le due righe seguenti dichiarano due handle ad un intero:

```
int^ intHandle;
int ^intHandle2;
```

Come si può notare un handle richiede un carattere ^ (si pronuncia hat) subito dopo il nome del tipo, oppure che preceda il nome della variabile.
I puntatori invece si dichiarano utilizzando il carattere *:

```
int* intPunt;
int *intPunt2;
```

È necessario prestare attenzione quando si dichiarano due o più handle o puntatori su una stessa riga:

```
String^ strHandle, strHandle2;
```

strHandle è un handle ad un oggetto *String*, mentre la seconda variabile *strHandle2*, nonostane il nome, non lo è. È soltanto un oggetto *String*. Per dichiarare corettamente due handle sulla stessa riga bisogna scrivere:

```
String ^strHandle, ^strHandle2;
```

INIZIALIZZAZIONE

Una volta dichiarata una variabile può essere inizializzata, cioè le si può dare un valore iniziale. Un modo per farlo è mediante una semplice assegnazione. Per esempio, data la variabile intera *i*, e la variabile double *d*, dichiarate come di seguito:

```
int i;
double d;
```

si può inizializzarle così:

```
i=0;
d=1.0;
```

Un'altra possibilità è la dichiarazione e l'inizializzazione contemporanea:

```
int i=0;
double d=1.0;
```

Se si hanno più dichiarazioni sulla stessa riga bisogna prestare ancora una volta attenzione:

```
int i=0, j=1;
```

Scrivendo infatti solo:

```
int i, j=1;
```

solo la variabile *j* verrà inizializzata a 1. Se si vuole inizializzarle entrambe bisognerà scrivere:

```
int i=j=1;
```

Infine, è possibile utilizzare la notazione funzionale:

```
int i(0);
double d(1.0);
```

BLOCCO DI ISTRUZIONI

Ogni dichiarazione deve essere univoca, cioè un identificatore deve essere distinto da ogni altro. Ma come possiamo essere sicuri che un identificatore, cioè una variabile, sia veramente unica? Immaginiamo ad esempio di mettere mano ad un programma scritto da qualcun altro, a questo punto sarebbe praticamente impossibile essere sicuri dell'univocità di un identificatore. Ed è a questo punto che entra in gioco il concetto fondamentale di blocco di codice e dei concetti di scope e durata di una variabile. Un blocco è una parte di codice delimitata da { e }, ed è all'interno di ogni blocco che le variabili devono avere un nome univoco. Il concetto di *scope*, o ambito di una variabile, è strettamente collegato al concetto di *blocco*, in quanto una variabile ha raggio d'azione che inizia dalla sua dichiarazione e termina alla fine del blocco in cui è stata dichiarata. È necessario sottolineare il fatto che ogni blocco può contenere dei sottoblocchi, cioè blocchi di codice annidati, all'interno dei quali le variabili dichiarate nei blocchi più esterni sono ancora in azione, cioè sono visibili.

```
class MainClass
{
    static void Main()
```

```
    {
        int a=10; //dichiarazione della variabile a

        {
            int b=a; //dichiaro b ed a è ancora visibile
            System.Console.WriteLine(b);
        }//fine del blocco annidato
        //qui b non è più visibile, la riga seguente darebbe errore
        //System.Console.WriteLine(b);

        System.Console.WriteLine(a); // a è ancora attiva
    }
}
```

In blocchi diversi è dunque lecito dichiarare variabili con lo stesso nome, in quanto lo scope delle variabili è diverso, ed in questa maniera si risolve il problema dell'unicità delle variabili. Il codice seguente è quindi perfettamente lecito:

```
{
    int var=10;
}  //qui termina l'ambito della prima var
{
    //in questo blocco dichiaro un'altra variabile var
    int var=5;
}
```

Ma anche un blocco annidato è un blocco diverso rispetto a quello che lo contiene, eppure non è possibile dichiarare una variabile con lo stesso nome di una esterna al blocco. Ad esempio, consideriamo il seguente codice:

```
public class EsempioScope
```

```
{
      static void Main()
      {
            int intero=0;
            while(intero<5)
            {
                  System.Console.WriteLine(intero);
                  intero=intero+1;
            }
      }
}
```

La variabile *intero* è dichiarata all'interno del blocco delimitato dal metodo *Main()*, all'interno del quale abbiamo un altro blocco, quello definito dall'istruzione while. La variabile intero in questo caso è visibile all'interno del while, cioè lo scope della variabile si estende nel blocco più interno.
Infatti se provassimo a dichiarare all'interno del while un'altra variabile con nome intero, otterremmo un'errore di compilazione del tipo:
error CS0136:
Una variabile locale denominata "intero" non può essere dichiarata in questo ambito perché darebbe un significato diverso a "intero", che è già utilizzato in un ambito "padre o corrente" per identificare qualcos'altro.
In generale, quindi, due variabili con lo stesso nome non possono essere dichiarate all'interno dello stesso blocco di codice o in blocchi che hanno stesso scope, come l'esempio visto a riguardo del *while*. L'unica eccezione si ha, come vedremo nel paragrafo seguente, quando abbiamo a che fare con variabili che siano campi di classe.

TIPI DI DATI

Tutti i tipi di dati in C++/CLI sono degli oggetti. I tipi di dati predefiniti pos-

sono essere suddivisi in due categorie principali: i tipi riferimento ed i tipi fondamentali. I tipi di dati fondamentali sono dei tipi che possono contenere un valore, memorizzandolo sullo stack, ma che in caso di necesità possono essere inscatolati all'interno di un oggetto, mediante la procedura detta boxing, e passando in questo caso in memoria heap. Al contrario, la procedura di unboxing può estrarre un tipo di dati fondamentale da un oggetto. I tipi riferimento sono invece sempre e comunque degli oggetti memorizzati nel managed heap.

Tipi fondamentali

Tutti i tipi standard di C++ sono utilizzabili in C++/CLI, ma in realtà, nonostante l'apparenza, non si tratta degli stessi tipi. Il Common Type System di .NET mette a disposizione tutti i tipi di dati equivalenti, e quindi scrivendo *int* o *double* in realtà si utilizzeranno degli alias dei tipi .NET corrispondenti, *System::Int32* e *System::Double*.

I tipi fondamentali possono essere raggruppati in 5 gruppi:

interi
floating-point
decimal
caratteri
booleani

I tipi interi sono *char*, *short*, *int*, *long* e i corrispondenti ottenuti anteponendo ad essi la parola chiave unsigned. Essi hanno dei corrispondenti tipi .NET *SByte*, *Int16*,*Int32*, *Int64* di .NET, e per i tipi unsigned *Byte*, *UInt16*,*UInt32*, *UInt64*.
I tipi interi permettono di rappresentare dei numeri interi, di dimensione variabile, a partire da un singolo byte, con il tipo *Byte* o *unsigned char*, ad arrivare ai 64 bit del tipo *Int64* o *long*.
I tipi a virgola mobile o floating point, sono due: *float* e *double*, che corrispondono in .NET ai tipi *Single* e *Double*. Essi permettono di rappre-

sentare numeri a virgola mobile rispettivamente di 32 bit a singola precisione, e 64 bit a doppia precisione.
Il tipo *Decimal* non esiste nel C++ standard, esso consente di scrivere numeri a 128 bit, con ben 28 cifre decimali, quindi utilizzabili per operazioni di alta precisione, ad esempio quando si ha a che fare con valute monetarie. Il tipo *wchar_t* o *System::Char* rappresenta un singolo carattere Unicode, a 16 bit. I programmatori C++ facciano attenzione a non confonderlo con il tipo *char*, che invece rappresenta un carattere ASCII a 8 bit. Per creare un carattere unicode basta utilizzare come prefisso una *L*. Per esempio scrivendo:

```
System::Char ch=L'a';
```

il carattere *ch* conterrà l'equivalente Unicode del carattere '*a*'.
Infine il tipo *bool* o *Boolean* è l'unico tipo booleano, che rappresenta il valore true o diverso da 0, oppure *false* o uguale a zero. In realtà il compilatore dara un warning se per inizializzare un tipo *bool* non si utilizza uno dei valori *true*, *false*, *1,0*.

Tipi riferimento

Un tipo *riferimento* può essere pensato come un handle ad un dato all'interno del *managed heap*. A differenza del puntatore classico, che punta nell'area di memoria heap nativa, gli indirizzi degli handle non possono essere manipolati, per esempio sommando degli offset.
A differenza dei tipi fondamentali dunque, i tipi riferimento non contengono un dato, ma sono un riferimento ad essi, un concetto equivalente al puntatore C/C++.
La Base Class Library del .NET Framework è praticamente composta da migliaia di tipi riferimento. Il tipo principale, la classe madre di tutti i tipo .NET è il tipo *Object*. Anche il tipo *String* in .NET è un tipo riferimento, e quindi ogni stringa viene allocata nel managed heap e referenziata mediante un handle.
È necessario quindi dichiarare una variabile stringa così:

```
String ^s="s";
```

mentre, scrivendo solo

```
String s="s";
```

il compilatore non gradirebbe.

TIPI PERSONALIZZATI

In C++/CLI il programmatore può creare i propri tipi di dati. Essi possono essere di due categorie: tipi valore che vengono allocati sullo stack, e tipi riferimento che invece vivono come già detto nell'heap gestito e che vengono ripuliti, al termine del loro ciclo di vita, dal Garbage Collector.

Tipi valore

Possono essere creati 3 tipi valore in C++/CLI:
value class
value struct
enum class / enum struct
Un oggetto *value class* è una struttura i cui membri hanno accesso privato, mentre un oggetto *value struct* è esattamente la stessa cosa ma i suoi membri hanno per default accesso pubblico.

```
value class VC
{
	int i;
};
value struct VS
{
	int i;
};
```

Dichiarando due variabili di tipo *VC* e *VS*, e tentando di inizializzare il membro *i* si avrà un comportamento diverso, secondo quanto detto:

```
VS vs;
vs.i=0; //OK

VC vc;
vc.i=0; //errore, il membro i è privato
```

È necessario quindi nel secondo caso creare un costruttore oppure dichiarare il campo intero i come *public*. Si suppinga di voler creare una classe *Punto*, che rappresenti un punt del piano cartesiano, con le coordinate *x,y*:

```
value class Punto
{
    int x,y;

public:
    Punto(int _x,int _y)
    {
        x=_x;
        y=_y;
    };
};
```

Per creare una istanza di *Punto* bisognerà fornire i valori di *x* e *y*:

```
Punto^ pt= Punto(1,2);
```

In questa maniera le due variabili membro *x,y*, private per default saranno inizializzate. Per creare una enumerazione basta invece racchiudere gli elementi fra parentesi graffe e separarli con la virgola. Per esempio

l'enumerazione dei giorni della settimana potrebbe essere scritta:

```
enum class GiorniSettimana { Lun, Mar, Mer, Gio,Ven,Sab,Dom  };
```

Per utilizzare l'enumerazione invece bisogna dichiarare un handle al tipo *GiorniSettimana*, ed inizializzarlo utilizzando uno dei valori:

```
GiorniSettimana giorno=GiorniSettimana::Lun;
```

Tipi riferimento

Si è già detto cos'è un tipo riferimento. Lo sviluppatore, oltre ad utilizzare i tipi riferimento forniti dal framework, può crearne di 4 tipologie differenti. Tali quattro tipologie sono le classi, le interfacce, i delegate, gli *array*.
Una volta creato un tipo riferimento, per utilizzarlo bisogna crearne una istanza, e per farlo ci si servirà dell'operatore *gcnew*. Si tornerà a parlare dei tipi riferimento nel capitolo dedicato alla programmazione object oriented in C++/CLI, per ora si darà qualche breve cenno per permettere al lettore meno esperto di seguire e di comprendere meglio i concetti introduttivi della programmazione.

L'operatore gcnew

L'operatore *gcnew* è il corrispondente dell'operatore *new*, che è appartenente al mondo delle classi native. Il nuovo operatore permette di creare una istanza di un tipo riferimento, e restituisce un handle ad essa. L'operatore classico *new* crea invece un puntatore ad un oggetto all'interno dell'heap CRT.
Riprendendo la value class Punto del paragrafo precedente, ecco come potrebbe essere creata un handle ad una istanza di essa utilizzando *gcnew*:

```
Punto^ pt=gcnew Punto(0,0);
```

C++/CLI introduce un nuovo valore chiamato *nullptr*, che rappresenta

un puntatore nullo oppure un valore nullo di handle. Il valore *nullptr* viene convertito implicitamente ad un puntatore, e viene valutato come 0, oppure ad handle, e viene valutato come un riferimento nullo, cioè a nessun oggetto in memoria.

Le classi

Una classe è un concetto fondamentale della programmazione ad oggetti. Sfogliando le prime pagine di un testo dedicato alla OOP (Object Oriented Programming), una frase tipica che vi si legge è che tutto è un oggetto. Citando una definizione più seria e precisa data da uno dei padri del paradigma stesso, Grady Booch, possiamo dire che un oggetto è un qualcosa che ha un suo stato, una sua identità, ed un suo comportamento. Dal punto di vista dello sviluppatore software dunque, un oggetto è un qualcosa che mantiene dei dati interni, che fornisce dei metodi per manipolarli, e che risiede in una propria riservata zona di memoria, convivendo con altri oggetti, dello stesso tipo o di altro tipo.
In Object Oriented, in generale, il tipo di un oggetto è la sua classe, cioè ogni oggetto appartiene ad una determinata classe. Ad esempio, un cane di nome *Argo*, che ha una propria identità, un proprio comportamento, ed uno stato (ad esempio il peso, l'altezza, la data di nascita), è un elemento di una classe, la classe *Cane*.
Il concetto di *classe* è per certi versi sinonimo a *tipo*. Ogni classe definisce le caratteristiche comuni di ogni oggetto che vi appartiene. Ad esempio tutti gli elementi della classe *Cane*, hanno quattro zampe, abbaiano, camminano. Ma ogni oggetto avrà poi delle caratteristiche che lo distinguono da ogni altro.
Ogni elemento di una classe, cioè ogni oggetto, si dice essere *istanza* di quella classe, e la creazione stessa di un oggetto viene anche detta, quindi, istanziazione dell'oggetto.
Tornando a parlare di sviluppo di software, quando programmiamo in maniera orientata agli oggetti, non dobbiamo fare altro che pensare ai concetti del dominio che stiamo affrontando, e ricavarne dunque gli oggetti e le funzionalità che devono avere e fornire, e le modalità di comunicazio-

ne fra oggetti diversi o della stessa classe.
Inoltre ogni oggetto, nel mondo comune, non è un entità atomica, cioè ogni oggetto è formato da altri oggetti, il classico mouse ad esempio è formato da una pallina, da due o più tasti, magari da una rotellina, tutti oggetti che aggregati forniscono la funzionalità totale del mouse.
Si tornerà a parlare presto di classi ed oggetti, per il momento basta tenere bene a mente che una classe è il building block fondamentale della maggior parte dei programmi C++/CLI, e che una classe è fatta da membri, proprietà e metodi.

Interfacce

Un'interfaccia può essere considerata lo schema di una classe, è come se fosse un contratto che una classe promette di mantenere.
Ogni interfaccia permette di definire le proprietà e i metodi che una classe che la implementa esporrà al suo utilizzatore.
Un'interfaccia non fornisce d'altra parte alcuna implementazione di tali metodi e proprietà, perché essa sarà invece fornita dalla classe.

Delegate ed eventi

Un *delegate* è un tipo riferimento che fornisce le stesse funzionalità note ai programmatori C++ come "puntatori a funzione".
In genere essi vengono utilizzati quando è necessario invocare una determinata funzione in risposta ad un evento, ed in maniera dinamica, cioè senza sapere a priori quale funzione dovrà essere eseguita.
Il concetto può sembrare un po' confuso ed astruso ai lettori alle prime armi, ma lo si chiarirà in seguito.
Un evento permette a un oggetto di scatenare l'esecuzione di un dato codice, rispondendo a qualcosa che è accaduta all'interno della classe in oggetto. In .NET un evento è inoltre una versione specializzata di delegate.

Array

Ogni linguaggio di programmazione (o quasi) possiede e implementa il-

concetto di array o vettore, cioè una sequenza di elementi. A differenza degli array classici di C o C++, in C++/CLI, e quindi generalmente in .NET, gli array sono sequenze di lunghezza fissa, che se violata genererà un'eccezione. Tutti gli array inoltre saranno derivati dalla classe madre *System::Array*, che fornirà numerosi metodi utili per la loro manipolazione ed utilizzo. Per comodità, è stata introdotta la parola chiave *array*, quindi per dichiarare e creare un vettore di interi, di lunghezza 5 sarà necessario innanzitutto dichiarare un handle alla parola chiave *array*, racchiudendo fra parentesi angolate il tipo degli elementi:

```
array<int>^ vettore;
```

per creare l'array, dato che esso sarà allocato nel managed heap, è necessario utilizzare l'operatore *gcnew*:

```
vettore=gcnew array<int>(5);
```

A questo punto, è possibile utilizzre l'array vettore mediante la classica notazione con le parentesi quadre. Per esempio per inserire il valore 1 alla prima posizione si dovrà scrivere così:

```
vettore[0]=1;
```

Si ricordi che gli array, cioè gli elementi che li costituiscono sono numerati a partire da 0 e non da 1.
È possibile dichiarare e creare array anche multidimensionali. Per esempio per creare una tabella o matrice, basterà creare un array di dimensione 2.

```
array<int,2>^ matrice = gcnew array<int,2>(3,3);
```

Per accedere alle singole celle della tabella ancora una volta basta indicare fra parentesi quadre l'indice di riga e colonna:

```
for(int i=0;i<3;i++)
{
	for(int j=0;j<3;j++)
	{
		matrice[i,j]=i*j;
	}
}
```

Il precedente frammento di codice riempie la tabella bidimensionale o matrice utilizzando due cicli *for* innestati. L'argomento array verrà approfondito nel proseguio.

Conversioni di tipi semplici

Quando si effettua un'assegnazione di variabile, può capitare che il tipo della variabile sia differente da quello del risultato di un'espressione da assegnare. Per esempio si supponga di avere una variabile intera e di voler assegnare il risultato di una operazione di divisione, che magari sia *double*.

```
int risultato= (123.1 / 4.3);
Console::Writeline(risultato);
```

Il compilatore in questo caso avvisa che nella conversione da *double*, risultato della divisione, a *int* ci potrebbe essere una perdita di dati, ed infatti il risultato stampato dalla successiva istruzione *WriteLine* è 28, che è un valore arrotondato.
È necessario, in questo e nei casi analoghi, una operazione detta conversione di tipo o *cast*.
Infatti la conversione automatica non è sempre l'operazione corretta, può capitare per esempio che il tipo della variabile abbia dimensione minore del risultato, e quindi non riesca a contenerlo. Il compilatore avvisa di ciò ma in questi casi è il programmatore che si deve assumere la responsabilità ed effetuare esplicitamente un cast del tipo. Per effettuare un cast

esplicito può essere utilizzato l'operatore di cast classico, indicando fra parentesi il tipo di destinazione, prima dell'espressione da convertire:

```
int i=(int)risultato;
```

oppure può essere utilizzata la sintassi seguente:

```
int i=safe_cast<int>(risultato);
```

In C++/CLI l'uso della sintassi classica, con le parentesi, fa in modo che prima di tutto venga tentato un *safe_cast*, quindi le due sintassi sono spesso equivalenti. Quando si hanno espressioni che coinvolgono più variabili di tipo diverso, è necessario che tutti i dati siano convertiti allo stesso tipo. Per esempio se si vuol sommare un intero con un *float*, allora tutti e due le variabili vengono convertite a *float*. Nella maggior parte dei casi queste conversioni sono automatiche, ma capita di dover agire manualmente scegliendo i tipi di destinazione.

```
double a = 1.2;
double b = 3.4;
int somma = a+b;
```

È dunque necessario effettuare due cast espliciti:

```
somma = (int) a + (int) b;
```

oppure il cast della somma ad intero:

```
somma=(int)(a+b);
```

Più avanti, dopo aver discusso dell classi, si vedrà come effettuare la conversione degli oggetti di tipo classe o struct.

CONTROLLO DI FLUSSO

Per controllo di flusso di un programma si intende quell'insieme di funzionalità e costrutti che un linguaggio mette a disposizione, per controllare l'ordine di esecuzione delle istruzioni del programma stesso. Se non potessimo controllare tale flusso, quindi, il programma sarebbe una sequenza di istruzioni, eseguite dalla prima all'ultima, nello stesso ordine in cui sono state scritte dal programmatore.

OPERATORI

Per controllare il flusso di un programma è necessario poter valutare delle espressioni. Le espressioni a loro volta sono scritte per mezzo di operatori applicati ad uno o più operandi. Una volta valutate, le espressioni restituiscono un valore, di un determinato tipo, oppure gli operatori possono variare il valore stesso di un operando, in questo caso si dice che essi hanno un side-effect, cioè effetto collaterale.
Il linguaggio C++/CLI fornisce tre tipi di operatori:

- **Unari**: sono operatori che agiscono su un solo operando, o in notazione prefissa, cioè con l'operatore che precede l'operando, ad esempio come l'operatore di negazione -*x*, o in notazione postfissa, in cui l'operatore segue un operando, come ad esempio l'operatore di incremento *x*++.
- **Binari**: sono operatori che agiscono su due operandi in notazione infissa, cioè l'operatore è frapposto agli operandi, come ad esempio il classico operatore aritmetico di somma *x+y*.
- **Ternari**: sono operatori che agiscono su tre operandi. L'unico operatore ternario è l'operatore condizione ?:, anch'esso in notazione infissa, *x* ? *y* : *z*, e di cui vedremo più avanti nel capitolo l'utilizzo.

In un'espressione gli operatori vengono valutati utilizzando ben precise regole di precedenza e di associatività. C++/CLI possiede diversi operatori per eseguire le operazioni aritmetiche fondamentali, per eseguire dei confronti e le classiche operazioni logiche booleane, delle assegnazioni, delle operazioni direttamente su dei bit, o degli operatori specializzati per trattare oggetti e i loro membri.

GLI OPERATORI ARITMETICI

Gli operatori aritmetici di C++/CLI sono quelli che permettono di eseguire le classiche operazioni aritmetiche sui tipi numerici, sia interi, sia a virgola mobile ed infine sui decimali. C++/CLI possiede in particolare sette operatori aritmetici. Oltre ai quattro classici operatori di somma +, sottrazione -, moltiplicazione *, e divisione /, C++/CLI mette a disposizione l'operatore modulo % che permette di calcolare il resto di una divisione fra interi:

```
int x=10;
int y=3;
int resto=x%y ; // restituisce 1
int quoz= x/y; //restituisce 3
```

Gli operatori di decremento e incremento permettono, come dice il nome, di incrementare o decrementare di una unità il valore di una variabile:

```
int x=0;
x++; // x vale ora 1;
x--; // x vale di nuovo 0
```

Ma la particolarità degli operatori di incremento e decremento è che essi possono essere usati sia in notazione prefissa, cioè con l'operatore che precede l'operando, sia in notazione postfissa come nell'esem-

pio sopra. Nel primo caso il valore dell'operando viene prima incrementato e poi viene restituito per essere utilizzato in un'espressione; nel secondo caso invece viene prima utilizzato il valore attuale, solo dopo esso viene incrementato. Vediamo un esempio che renderà chiare le differenze nell'utilizzo delle due notazioni.

```
Console::WriteLine("i: {0}", i);
Console::WriteLine("++i: {0}", ++i); //Pre-incremento
Console::WriteLine("i++: {0}", i++); //Post-incremento
Console::WriteLine("i: {0}", i);
Console::WriteLine("--i: {0}", --i); //Pre-decremento
Console::WriteLine("i--: {0}", i--); //Post-decremento
Console::WriteLine("i: {0}", i);
```

Se provate ad eseguire il codice precedente, otterrete le seguenti righe di output:

```
i: 0
++i: 1
i++: 1
i: 2
--i: 1
i--: 1
i: 0
```

Nella prima riga di codice viene inizializzata la variabile i, ed il suo valore viene stampato nella seconda linea, nella terza la variabile i viene incrementata e solo dopo l'incremento viene usato il suo valore per stamparlo, che infatti è adesso pari a 1, nel caso del post-incremento invece viene prima stampato il valore attuale di i, ancora 1, e poi viene incrementato, infatti nella linea successiva il valore di i sarà 2. Allo stesso modo si spiegano i due successivi decrementi.

OPERATORI DI CONFRONTO

Gli operatori di confronto sono fondamentali per controllare il flusso del programma. Essi restituiscono un valore booleano e dunque vengono utilizzati per confrontare i valori di due variabili o in generale di due espressioni. Gli operatori relazionali supportati da C++/CLI sono < (minore di), > (maggiore di), <= (minore o uguale), >= (maggiore o uguale), == (uguale a) e != (diverso da).
Il codice seguente illustra l'utilizzo di tutti gli operatori relazionali, che funzionano con tutti i tipi predefiniti del linguaggio:

```
bool b;
int cento=100;
int zero=0;
b=(cento>zero);
Console::WriteLine("100 > 0 ? {0}",b);
b=(cento<zero);
Console::WriteLine("100 < 0 ? {0}",b);
b=(cento==zero);
Console::WriteLine(cento +"== " + zero +"?" + b);
b=(cento>=100);
Console::WriteLine(cento +">=100 ?" + b);
b=(cento<=100);
Console::WriteLine(cento +"<=100 ?" + b);
b=(cento!=100);
Console::WriteLine(cento +" != 100 ?" + b);
b=(cento!=0);
Console.WriteLine(cento +" != 0 ?" + b);
```

Il programma darà in output il risultato di tutti i confronti effettuati fra due variabili:

```
100 > 0 ? True
100 < 0 ? False
```

```
100== 0? False
100>=100 ? True
100<=100 ? True
100 != 100 ? False
100 != 0 ? True
```

Per il tipo predefinito *String* sono definiti gli operatori == e !=, che permettono di verificare l'uguaglianza o meno di due stringhe. Due stringhe sono considerate uguali se si verifica uno dei seguenti casi: entrambe sono *null* oppure hanno uguale lunghezza (cioè stesso numero di caratteri) e i caratteri che le compongono sono uguali in ogni posizione. In parole povere:

```
String^ uomo="antonio";
String^ donna="caterina";
bool b=(uomo==donna); // restituisce false
b=(uomo=="antonio"); //è true
```

Operatori logici

Gli operatori logici sono simili ai precedenti operatori di confronto, ma agiscono su operandi di tipo booleano. Esistono i tre seguenti operatori:

- ! è l'operatore *NOT*: se l'operando è true esso restituisce il valore *false* e viceversa.
- && è l'operatore *AND*: se entrambi gli operandi sono *true*, restituisce true, altrimenti *false*.
- || è l'operatore *OR*: se almeno un operando è *true*, esso restituisce true, altrimenti *false*.

Operatori bitwise

Gli operatori bitwise o bit a bit, agiscono sui singoli bit degli operandi che devono essere degli interi. Esistono 4 tipi di operatori bitwise (più i due operatori di shift):

AND: &
OR: |
XOR: ^
Complemento a uno o di inversione: ~

Gli operatori logici bitwise eseguono le operazioni booleane sui bit di pari posto dei due operandi, quindi per utilizzarli con profitto, come vedremo anche per gli operatori di Shift, bisogna conoscere un po' di numerazione binaria. L'operatore binario & esegue un *AND logico* fra i bit di due operatori, vale a dire che per ogni coppia di bit restituisce un bit 1 se essi sono entrambi pari ad 1:

```
int x=5;//in binario 0000 0000 0000 0000 0000 0000 0000 0101
int y=9;//in binario 0000 0000 0000 0000 0000 0000 0000 1001
int z=x & y; // restituisce il valore 1
```

infatti l'unica coppia di bit pari a 1 è quella in posizione 0, e dunque il risultato sarà un unico 1 in posizione 0 e tutti i restanti bit pari a zero, in decimale 1. Analogamente, l'operatore | effettua l'*OR logico* degli operandi, quindi da un 1 se almeno un bit della coppia è pari ad 1:

```
z = x | y ; // restituisce 13
```

Mentre l'operatore ^ è l'operatore di *OR esclusivo* (o *XOR*) che da un 1 per ogni coppia in cui c'è uno ed un solo bit 1.

```
z = x ^ y; // restituisce 12
```

L'unico operatore unario è l'operatore ~ (tilde) di inversione bitwise, che inverte tutti i bit di un operando.

```
unsigned char ux=10;//in binario 0000 1010
```

```
unsigned char uy=~ux;//in 1111 1010 = 245
Console::WriteLine(uy);
```

È naturalmente necessario conoscere il sistema di numerazione binario per comprendere le conversioni da binario a decimale e viceversa degli esempi precedenti.

OPERATORI DI SHIFT

Gli operatori di shift eseguono, come i precedenti, delle operazioni sui bit di un operando. L'operatore << è l'operatore di *shift* (scorrimento) a sinistra, cioè scorre tutti i bit dell'operando a sinistra di tante posizioni quanto indicato dall'operando destro:

```
int x=1; //in binario 0000 0000 0000 0000 0000 0000 0000
                                                      0001
int y=x<<2 // 0000 0000 0000 0000 0000 0000 0100 = 4
```

L'operatore >> effettua invece lo *shift* a *destra*, inserendo a sinistra dei bit, con la cosiddetta estensione di segno: nel caso in cui il valore dell'operando da scorrere è positivo vengono inseriti dei bit 0, in caso contrario, valore negativo, vengono inseriti degli 1.

```
int x=15; //in binario 0000 0000 0000 0000 0000 0000 0000
                                                      1111
int y=x>>2; // 0000 0000 0000 0000 0000 0000 0000 0011 =
3
x=-4; // 1111 1111 1111 1111 1111 1111 1111 1100
y=x>>2; // 1111 1111 1111 1111 1111 1111 1111 1111 = -1
```

L'operatore condizionale

L'operatore condizionale è l'unico operatore ternario di C++/CLI. Esso usa quindi tre espressioni.

L'operatore prende la prima espressione valutando se essa sia *true* o *false*. Nel primo caso, viene eseguita la seconda espressione, quella subito dopo il ?, viceversa verrà eseguita la terza, vale a dire quella che segue i due punti:

```
espressione1 ? espressione2 : espressione3;
```

Esso è come un modo di abbreviare il seguente costrutto *if/else* (vedi più avanti l'istruzione *if*):

```
if(expr1)
return expr2;
else return expr3
```

Per avere un esempio, ecco come è possibile calcolare il massimo fra due numeri utilizzando solo l'operatore ternario:

```
//operatore condizionale
int x=1;
int y=2;
int max;
(x>y)? max=x : max=y ;
Console::WriteLine(max);
```

Molti sviluppatori preferiscono evitare l'utilizzo dell'operatore ternario, così come molte aziende che sviluppano software ne proibiscono l'utilizzo nei propri documenti di stile a causa della presunta oscurità del codice che ne deriva. Al contrario, a mio modesto avviso, è innegabile la sua eleganza ed espressività. Comunque, per tagliare la testa al toro, direi che è perlomeno necessario comprenderne l'uso e la semantica nell'eventualità di incontrarlo nel codice scritto da altri.

Operatori di assegnamento

Esistono undici operatori di assegnamento che i programmatori C++/CLI possono utilizzare, esposti qui di seguito. L'operatore utilizzato per assegnare un valore a una qualsiasi variabile di un qualunque tipo, operazione fondamentale in qualunque programma, è il segno di uguale (=). L'espressione che si trova alla destra del segno viene calcolata, il risultato viene assegnato alla variabile o all'operando che si trova a sinistra. Per esempio:

```
y=2;
x=y+1;
```

Le istruzioni precedenti prima assegnano il valore 2 alla variabile y. Poi viene assegnato alla variabile *x* il valore 3, risultato dell'espressione *y+1*. È possibile utilizzare gli altri operatori di assegnazione, detti anche di assegnazione composta, per eseguire un calcolo e contemporaneamente assegnare il risultato ad una variabile. In generale un operatore di assegnazione composta assume la forma op =, l'espressione:

```
x op= y;
```

è equivalente a scrivere la seguente assegnazione:

```
x= x op y;
```

Gli operatori di assegnazione composta sono i seguenti:

-= sottrae l'espressione e poi assegna il risultato
*= moltiplica e poi assegna
/= divide e poi assegna
%= calcola il resto e poi assegna
>>= shift a destre e poi assegna

<<= shift a sinistra e poi assegna
&= AND bitwise e poi assegna
^= XOR bitwise e poi assegna
|= OR bitwise e poi assegna

Quindi scrivere *x -= 1* è un'abbreviazione di *x=x-1*.

Operatori di indirizzamento

Per gestire handle e puntatori in C++/CLI è possibile utilizzare tre differenti operatori, tutti unari.
I programmatori C e C++ sanno bene che per ottenere l'indirizzo di un'oggetto o di una variabile si utilizza l'operatore di indirizzamento &:

```
int* x= &y; //assegna a x l'indirizzo in memoria di y
```

Se invece si vuol ottenere il contenuto di un indirizzo di memoria si utilizza l'operatore *, per esempio se *x* si trova all'indirizzo 100 in memoria, per inserire un valore a tale indirizzo, basta scrivere:

```
*x=1; //inserisce il valore 1 nell'indirizzo puntato da x
```

L'operatore di riferimento è stato introdotto invece con C++/CLI in maniera da avere un operatore analogo all'operatore di indirizzamento, ma che agisca sugli handle, cioè sugli oggetti .NET.
Per esempio:

```
int x=1;
int %rifX=x;
```

rifX è un riferimento ad un *int*. Se ora si assegna un valore a *rifX*, cambierà il valore a cui *rifX* si riferisce, cioè *x*;

```
rifX=2;
Console::WriteLine(x); // x vale 2
```

ISTRUZIONI DI SELEZIONE

C++/CLI fornisce due costrutti di selezione, cioè due costrutti per controllare il flusso di un programma selezionando quale dovrà essere la prossima istruzione o blocco da eseguire, selezione che si effettua in base al verificarsi di certe condizioni. Questi due costrutti, descritti nel seguito del paragrafo sono l'*if/else* e lo *switch*.

Il costrutto if/else

Il modo più comune per controllare il flusso di esecuzione di un programma è quello di utilizzare il costrutto di selezione condizionale *if/else*.
Il costrutto *if/else* può essere utilizzato in due maniere.
Nella prima, l'*if* è seguito da una condizione booleana racchiusa fra parentesi, e solo se essa è vera, quindi solo se assume valore *true*, viene eseguito il blocco successivo.

```
if( espressione_booleana )
    Blocco_istruzioni
```

Il blocco di istruzioni può essere sia una istruzione semplice seguita da un punto e virgola oppure un blocco di istruzioni racchiuso fra parentesi graffe.
Il secondo modo è quello che prevede l'utilizzo della parola chiave *else*. Se l'espressione booleana dell'*if* non è vera, verrà eseguito il blocco o l'istruzione semplice che segue l'*else*.

```
if(espressione_booleana)
    ramo_if
else
    ramo_else
```

I costrutti *if/else* possono essere annidati, cioè all'interno di un ramo *if* o di un ramo *else* possono essere inseriti altri costrutti *if* o *if/else*. In tali casi è opportuno ricordare che l'*else* si riferisce sempre all'*if* immediatamente precedente, a meno che non si utilizzino le parentesi graffe per delimitare i blocchi. Ad esempio il seguente blocco di istruzioni, fra l'altro indentato male, potrebbe far credere che il ramo else sia riferito al primo *if*:

```
if(x==0)
    if(y==0) Console::WriteLine("ramo if");
else Console::WriteLine("ramo else");
```

invece tale codice è equivalente al seguente, in quanto l'*else* si riferisce come detto prima, all'*if* immediatamente precedente:

```
if(x==0){
    if(y==0){
        Console.WriteLine("ramo if");
    }
    else{
        Console.WriteLine("ramo else");
    }
}
```

In C++/CLI è possibile utilizzare come condizione dell'*if* un'espressione di assegnazione, ad es. *if(x=0)*, in quanto in tale linguaggio è possibile usare come condizione il fatto che una variabile intera sia nulla o meno. Ciò spesso porta ad errori di codice difficilmente individuabili:

```
int x=0;
if(x=1)
Console::WriteLine("x=1");
```

Ad un occhio poco esporto potrebbe sfuggire il fatto che l'espressione condizionale dell'*if* è in realtà una semplice assegnazione della variabile x, e quindi l'istruzione WriteLine viene eseguita in ogni caso. La condizione di un costrutto *if* può essere naturalmente molto più complessa di una semplice espressione, ed implicare l'utilizzo combinato di più operatori logici booleani. Per esempio volendo eseguire una data istruzioni solo se due espressioni *a* e *b* sono contemporaneamente vere, si può scrivere:

```
if(a && b)
{
    …
}
```

Da notare che l'operatore &&, così come gli altri operatori logici, subisce una cosiddetta operazione di cortocircuito: se a è false, è inutile valutare b, tanto l'espressione complessiva di *AND* sarà in ogni caso *false*.

L'istruzione switch

Se è necessario effettuare una scelta fra diversi alternativi percorsi di esecuzione, e tale scelta è basata sul valore assunto a runtime da una variabile, allora è possibile usare il costrutto *switch*. La sintassi dello *switch* è la seguente:

```
switch(espressione)
{
    case costante1:
        blocco_istruzioni
        istruzione_salto;
    case costante2:
        blocco_istruzioni
        istruzione_salto;
```

```
    …
    [default:blocco_istruzioni]
}
```

L'espressione all'interno delle parentesi viene valutata, e se il suo valore corrisponde ad una delle costanti dei rami case, il flusso del programma salta al caso corrispondente, del quale viene eseguito il blocco di istruzioni, fino a raggiungere una necessaria istruzione di salto. Se è presente una label default, e la valutazione dell'espressione produce un risultato non contemplato dalle costanti dei case, il flusso va alla label di default, se tale label invece non è presente il flusso va alla fine dello *switch*.
Tipicamente, l'istruzione di salto che chiude un blocco case è l'istruzione, che fa uscire il programma dal costrutto switch. Ma è possibile utilizzare anche una istruzione di salto qualunque, tipo un return.
In C++/CLI è possibile omettere l'istruzione di salto, cioè è possibile, in uno *switch*, passare da una sezione case alla seguente:

```
switch(x)
{
    case 0:
        Console::WriteLine("x è nulla");
    case 1:
        Console::WriteLine("x è 0 oppure 1");
        break;
    default:
        Console::WriteLine("x è diverso da 0 e 1");
        break;
}
```

Si faccia attenzione quindi, la dimenticanza di un *break* può provocare malfunzionamenti difficilmente rintracciabili.

ISTRUZIONI DI ITERAZIONE

Le istruzioni di iterazione hanno lo scopo di eseguire ripetutamente un determinato blocco di istruzioni, formato eventualmente anche da una sola istruzione. Esse dunque permettono di eseguire più cicli attraverso uno stesso blocco di codice: un tipico esempio è quello in cui si esegue un ciclo per leggere o scrivere i diversi elementi di un vettore.

Il ciclo while

Il costrutto *while* permette di eseguire un blocco di istruzioni zero o più volte in base al valore di un'espressione booleana:

```
while(espressione_booleana)
      blocco_istruzioni
```

Fino a quando l'espressione restituisce valore *true*, il blocco istruzioni interno al ciclo *while* verrà eseguito. È necessario, per uscire dal ciclo *while*, che l'espressione ad un certo punto diventi false, oppure che all'interno del ciclo *while* ci sia un'istruzione di salto, che faccia passare l'esecuzione all'esterno o alla fine del ciclo *while*, cioè all'istruzione immediatamente successiva.

```
int i=0;
while(i<10)
{
      Console.WriteLine{"i={0}",i);
      i++;
}
```

L'esempio precedente esegue il blocco fra parentesi graffe fino a che i non sia pari a 10, per tale valore infatti l'espressione *i<10* diventa falsa.
Come detto prima è possibile anche terminare il ciclo con un break,

dunque anche da un ciclo while la cui espressione di valutazione sia sempre true, come nel caso seguente, è possibile uscire.

```
int i=0;
while(true)
{
      Console.WriteLine{"i={0}",i);
      i++;
      if(i=10) break;
}
```

Non sono rari i casi in cui, a causa di un errore di programmazione, il programma entra in un ciclo infinito, e come conseguenza sembra bloccato nella sua esecuzione. È bene assicurarsi dunque che da un ciclo ad un certo punto si esca, dando magari qualche feedback all'utente nel caso di cicli particolarmente lunghi, ad esempio con una *ProgressBar* se si tratta di un programma Windows.
È possibile utilizzare all'interno del ciclo while un'istruzione di salto *continue*, con la quale il blocco rimanente di istruzioni viene saltato e si ricomincia un nuovo ciclo rivalutando l'espressione booleana.

```
int i=0;
while(i++<10)
{
      if(i%2==0)
            continue;
      Console.WriteLine("i={0}",i);
}
```

L'esempio precedente stampa il valore di *i* se esso è dispari, altrimenti incrementa il valore della variabile i e ricomincia il ciclo.
Notate che se l'espressione booleana è *false* in partenza, il blocco di istruzioni del ciclo *while* non verrà mai eseguito. Ciò costituisce la

differenza principale con il ciclo *do…while* esposto nel successivo paragrafo.

Il ciclo *do while*

Il ciclo *do while* è analogo al precedente, con l'unica differenza che il blocco di istruzioni incorporato viene eseguito almeno una volta, in quanto l'espressione booleana viene valutata alla fine del primo ciclo. A questo punto, se essa assume valore true, si esegue un nuovo ciclo altrimenti si esce dal ciclo.

```
int i=10;
do{
    Console__WriteLine("i={0}",i++);
}while(i<10);
```

Il valore di *i* è 10 in partenza , ciò nonostante il ciclo viene eseguito ugualmente e stampa il valore di *i*.

Il ciclo *for*

I due cicli visti negli esempi precedenti, permettono di eseguire dei blocchi di istruzioni incrementando di volta in volta il valore di una variabile. Tale situazione è così frequente in un programma, che la maggior parte dei linguaggi di programmazione, e fra questi C++/CLI, fornisce un costrutto dedicato a tale funzionalità. Tale costrutto è il ciclo *for*. Il ciclo *for* ha la seguente sintassi generale:

```
for(inizializzazione; espressione_booleana; incremento)
    blocco_istruzioni
```

All'interno delle parentesi tonde che seguono la parola chiave *for*, vengono dunque eseguite tre diverse operazioni, tutte quante comunque facoltative. Come prima operazione, viene eseguita un'inizializzazione, in genere ad una variabile contatore di ciclo viene assegna-

to un valore iniziale, quindi viene valutata una condizione booleana, e se essa assume valore true viene eseguito il blocco di istruzioni. Alla fine di ogni iterazione viene eseguita la terza operazione, in genere un incremento della variabile contatore. Quando l'espressione diviene *false*, l'esecuzione del programma salta all'istruzione che segue il ciclo *for*.
Il ciclo *for* viene in genere impiegato per eseguire cicli che hanno un numero prestabilito di passi, ad esempio iterare lungo ogni elemento di un vettore, la cui lunghezza è nota o comunque calcolabile prima del ciclo:

```
array<int>^ vett=gcnew array<int>(5);
for(int i=0;i<vett->Length;i++)
{
      vett[i]=i;
      Console::WriteLine("vettore[{0}]={1}",i,vett[i]);
}
```

Tramite l'operatore virgola "," è possibile inoltre inizializzare e incrementare diverse variabili all'interno di un ciclo *for*, con la limitazione che tutte abbiano uguale tipo. Il metodo seguente crea una matrice identica di dimensione pari al parametro *dim*, cioè una matrice con tutti gli elementi nulli, tranne quelli sulla diagonale che saranno pari a 1:

```
array<int,2>^ matrice=gcnew array<int,2>(dim,dim);
for(int i=0,j=0;i<dim && j<dim;i++,j++)
{
      matrice[i,j]=(i==j)? 1: 0;
}
return matrice;
```

È naturalmente possible innestare più cicli *for*, ad esempio per trat-

tare con le matrici è naturale innestarne due, ognuno dei quali itera uno dei due indici delle matrici. Il seguente esempio esegue la somma di due matrici identità, create con il precedente metodo:

```
array<int,2>^ matriceA=CreaMatriceIdentita(3);
array<int,2>^ matriceB=CreaMatriceIdentita(3);
array<int,2>^ matriceC=gcnew array<int,2>(3,3);
for(int i=0;i<3;i++)
{
	for(int j=0;j<3;j++)
	{
		matriceC[i,j]=matriceA[i,j]+matriceB[i,j];
	}
}
```

Come già detto, le istruzioni fra le parentesi tonde del *for* sono tutte quante facoltative. Ad esempio è possibile implementare un ciclo infinito analogo ad un *while(true){…}*, scrivendo un ciclo *for* come il seguente:

```
for(;;)
{
	faQualcosa();
	break;
}
```

L'istruzione foreach

Visual Basic e C# forniscono un'istruzione che permette di scorrere gli elementi di collezioni o array senza far uso di espressioni booleane, ma in maniera più naturale, scorrendo gli elementi in modo sequenziale. Tale istruzione, mancante in C++, è stata introdotta anche in C++/CLI, ed è for each. La sintassi generale dell'istruzione *for each* è la seguente:

```
for each(tipo elem in collezione)
      blocco_istruzioni
```

Come esempio scriviamo il codice per la stampa degli elementi di una matrice, utilizzando un *for* e poi con un *for each*:

```
for(int i=0;i<3;i++)
{
      for(int j=0;j<3;j++)
      {
            Console::WriteLine(matrice[i,j]);
      }
}
```

Con un for each invece:

```
for each(int elem in matriceC)
{
      Console::WriteLine(elem);
}
```

Come è possibile notare l'istruzione *for each* fornisce maggiore chiarezza al codice, soprattutto quando si ha a che fare con delle collezioni di oggetti un po' più complesse rispetto ad un semplice array di interi, e quando quindi è necessario invocare dei metodi o delle proprietà dell'oggetto stesso. Il rovescio della medaglia si ha nel fatto che l'istruzione *for each* fornisce un accesso agli elementi della collezione in sola lettura, cioè nel suo blocco di istruzioni non è possibile modificare l'elemento a cui si sta accedendo.

ISTRUZIONI DI SALTO

Le istruzioni di salto permettono di trasferire il controllo dal punto in

cui esse vengono chiamate ad un altro punto del programma. Il punto di arrivo è detto target del salto. In genere l'utilizzo delle istruzioni di salto avviene per uscire da uno dei precedenti costrutti iterativi.

break

Si è già incontrata l'istruzione break nei paragrafi precedenti. Il break serve a far uscire il programma da uno dei blocchi di istruzione dei costrutti switch, *while*, *do*, *for* o *for each*, spostando l'esecuzione all'istruzione immediatamente successiva al blocco stesso.

```
int i=0;
while(true)
{
    if(i==10)
        break; //se i è 10 esce dal ciclo
    else Console::WriteLine(i++);
}
```

Nel caso in cui ci siano più blocchi innestati, ad esempio due cicli for, l'istruzione *break* si riferisce solo al ciclo in cui è presente, e fa dunque saltare l'esecuzione alla fine di tale ciclo. Per far uscire il programma ad un ciclo più esterno è possibile usare l'istruzione di salto incondizionato goto (vedi il paragrafo relativo).

```
int i=0;
for(int i=0;i<10;i++)
{
    for(j=0;j<10;j++)
    {
        if(j==i)
            break; //se i==j esce dal for interno
    }
```

```
}
```

Continue

L'istruzione *continue*, piuttosto che terminare il ciclo in cui si trova (ciclo che può essere *while*, *do*, *for* e *for each*), fa iniziare immediatamente una nuova iterazione, saltando la parte di ciclo che si trova dopo l'istruzione stessa. Riprendendo l'esempio già visto nel paragrafo dedicato al ciclo *while*:

```
while(i++<10)
{
    if(i%2==0)
        continue;
    Console::WriteLine("i={0}",i);
}
```

Come per l'istruzione *break*, l'istruzione *continue* si riferisce sempre al ciclo in cui l'istruzione *continue* stessa è richiamata, con la possibilità ancora di usare il *goto* per saltare più di un livello di annodamento.

Return

L'istruzione *return* è un'istruzione di salto che restituisce immediatamente il controllo al metodo che sta chiamando quello in cui si trova l'istruzione *return* stessa.

```
public void Primo()
{
    Secondo();
    //punto di ritono
}
…
```

```
public void Secondo()
{
      //fa qualcosa
      if(condizione)
            return;
      //fa qualcos'altro
}
```

Nell'esempio il metodo *Primo* chiama il metodo *Secondo*, nel quale se si verifica una certa condizione, il controllo viene passato di nuovo a *Primo*, esattamente nel punto successivo alla chiamata *Secondo()*.
La stessa cosa avviene nel caso in cui si raggiunga la fine del metodo *Secondo().*
Nel caso in cui il metodo restituisca un qualche valore è necessario utilizzare l'istruzione *return* seguita da una espressione che restituisce un valore del tipo dichiarato dal metodo stesso.

Goto

C++/CLI mette a disposizione la famigerata istruzione di salto incondizionato, l'istruzione *goto*!
Inizio subito col dire che esistono teoremi che provano il fatto che ogni programma è codificabile senza fare uso del *goto* o chi per lui (v. Jacopini-Bohem), e che illustri autori ne deplorano l'utilizzo se non addirittura l'esistenza stessa (v. Tanenbaum). Partendo da tali presupposti, con cui non posso che essere d'accordo, espongo comunque l'utilizzo dell'istruzione *goto*.

```
x=0;
      while(1)
      {
            if(x++>10)
                  goto LABEL1;
```

```
    }
LABEL1:
    Console::WriteLine("fine ciclo");
```

L'istruzione serve ad effettuare un salto incondizionato ad una riga contente la particolare *Label* di destinazione.

GLI ARRAY

Un array (o vettore) è una sequenza ordinata di un dato tipo. La posizione di ogni elemento è fissa, e viene detta indice dell'elemento. Gli array di C++/CLI, detti anche *managed array*, vengono allocati all'interno dell'area di memoria detta CLI Heap, e dunque sono classici oggetti di tipo riferimento.
Ogni array ha un *rank* (o dimensione), che determina il numero di indici che viene associato ad ogni elemento dell'array. Per esempi,o un array di dimensione 1, è un classico vettore di elementi, detto anche array monodimensionale. Un array con rank maggiore di 1 è invece detto multidimensionale.
Esiste anche un secondo tipo di array a più dimensioni, detto *jagged array*, su cui torneremo a breve.
La sintassi utilizzata per dichiarare un array in C++/CLI è stata già introdotta, basta utilizzare la parola chiave *array*, indicare il tipo degli elementi, ed il nome da dare alla variabile:

```
array<int>^ vettore;
```

Si noti che essendo come detto gli array deigli oggetti, vengono dichiarati come handle. Si faccia attenzione a distinguere il tipo dell'array dal tipo degli elementi, in questo caso per esempio gli elementi sono degli interi, dunque di tipo valore, mentre l'array è comunque di tipo riferimento. Se invece gli elementi dell'array fossero anch'essi di un tipo riferimento, è logico che bisogna utilizzare l'operatore ^

all'interno delle parentesi angolate:

```
array<String^>^ vettoreStringhe;
```

Si è più volte detto che ogni array è un oggetto di tipo riferimento, precisiamo che ogni array deriva dalla classe *System::Array*, e dunque sarà possibile utilizzarne i metodi.

Array monodimensionali

Gli array monodimensionali sono array con rank pari a 1. La sintassi per inizializzare un array a singola dimensione prevede di passare il numero di elementi che esso conterrà al costruttore, il seguente esempio crea un vettore di 10 elementi e ne inizializza ogni elemento:

```
array<int>^ intarr = gcnew array<int>(10);
for(int i=0;i<10;i++)
intarr[i]=i;
```

Una seconda possibilità prevede la dichiarazione e la contestuale inizializzazione degli elementi:

```
array<int>^ arr2=gcnew array<int>{1,2,3,4};
```

La classe *System.Array* fornisce parecchi metodi per lavorare con gli array, ed essendo ogni array derivato da essa possiamo per esempio conoscere la lunghezza di un array usando la proprietà *Length*:

```
Console::WriteLine("elementi dell'array {0}",intarr->Length);
```

Notate che una volta inizializzato, non si può variare la lunghezza dell'array, di conseguenza anche la proprietà *Length* è di sola lettura. Qualora si avesse la necessità di utilizzare vettori di lunghezza e dimen-

sione variabile la base class library .NET fornisce parecchie classi adatte a tale scopo.

Array multidimensionali

Se un array ha rank maggiore di uno, viene detto array multidimensionale. Si noti che in questo caso non si parla di *array di array* (detti invece array *jagged*). Il concetto di array multidimensionale è facilmente comprensibile se si pensa a quelli di dimensione due. In questo caso si ha infatti una tabella o matrice, cioè una struttura a più righe e più colonne. Per creare un array multidimensionale, il rank viene specificato subito dopo il tipo degli elementi, all'interno delle parentesi angolate, mentre la lunghezza di ogni dimensione viene specificata come parametri del costruttore. Il seguente esempio crea un array di stringhe di dimensione 2, con 3 righe e 2 colonne:

```
array<String^, 2>^ rubrica = gcnew array<String^, 2>(3,2);
```

Il nome dell'array è *rubrica*, quindi servirà a memorizzare nella prima colonna di ogni riga un nome, e nella seconda, il corrispondente numero telefonico:

```
rubrica[0,0] = "Antonio";
rubrica[0,1] = "390 5634982";
rubrica[1,0] = "Caterina";
rubrica[1,1] = "300 7341092";
rubrica[2,0] = "Polizia";
rubrica[2,1] = "113";
```

La consueta sintassi con le parentesi quadre permette di accedere agli elementi, ed in questo caso bisogna utilizzare due indici, uno relativo alla riga ed uno alla colonna, ecco come stampare tutti gli elementi, riga per riga:

```
Console::WriteLine("Nome\tTelefono");
Console::WriteLine();
for(int i=0; i<3; i++)
{
    for(int j=0; j<2; j++)
        Console::Write("{0}\t", rubrica[i,j]);
    Console::WriteLine();
}
```

Anche per gli array a più dimensioni è possibile dichiarare e inizializzare contemporaneamente. L'esempio precedente può essere riscritto così:

```
array<String^, 2>^ employees =
{
{"Antonio","390 5634982"},
{"Caterina","300 7341092"},
{"Polizia","LA"}
};
```

Array jagged

Gli array *jagged*, sono dei particolari tipi di array a più dimensioni, che però non sono da confondere con gli array multidimensionali. Essi sono in pratica vettori di vettori e possono essere non rettangolari, vale a dire, facendo sempre l'esempio a due dimensioni, un array di tale tipo ha ancora più righe ma ognuna di esse può avere diverso numero di colonne:

[1]
[2 3]
[4 5 6]

Per dichiarare ed inizializzare un array jagged, bisogna quindi crea-

re un array classico,e come tipo degli elementi inserire un altro array:

```
array<array<int>^>^ jagarr = gcnew array<array<int>^> (3);
```

L'array *jagarr* ha dimensione 1, e lunghezza 3, ed ogni elemento dell'array è a sua volta un array di interi. Ognuno degli array di interi deve essere a sua volta inizializzato:

```
for(int i=0;i<jagarr->Lenght;i++)
{
	jagarr[i]=gcnew array<int>(i+1);
}
```

PROGRAMMAZIONE AD OGGETTI

Questo capitolo intende fornire una rapida introduzione al paradigma di programmazione Object Oriented, dando una rapida infarinatura dei concetti principali, e soffermandosi su come essi sono visti da C++/CLI, e quindi sul concetto di ref class, che è un punto chiave della programmazione ad oggetti in C++/CLI.

OGGETTI E CLASSI

Cos'è un oggetto? Sfogliando le prime pagine della quasi totalità dei testi dedicati alla OOP (Object Oriented Programming), una frase tipica che vi si legge è che tutto è un oggetto. Citando una definizione più seria e precisa data da uno dei padri del paradigma stesso, Grady Booch, possiamo dire che un oggetto è un qualcosa che ha un suo stato, una sua identità ed un suo comportamento. Dal punto di vista dello sviluppatore software, dunque, un oggetto è un qualcosa che mantiene dei dati interni, che fornisce dei metodi per manipolarli, e che risiede in una propria riservata zona di memoria, convivendo con altri oggetti, dello stesso tipo o di altro tipo. In Object Oriented, in generale, il tipo di un oggetto è la sua classe, cioè ogni oggetto appartiene ad una determinata classe. Ad esempio un cane di nome *Argo*, che ha una propria identità, un proprio comportamento, ed uno stato (ad esempio il peso, l'altezza, la data di nascita), è un elemento di una classe, la classe *Cane*.

Il concetto di classe è sinonimo dunque di tipo. Ogni classe definisce le caratteristiche comuni di ogni oggetto che vi appartiene. Ad esempio, tutti gli elementi della classe *Cane*, hanno quattro zampe, abbaiano, camminano. Ma ogni oggetto avrà poi delle caratteristiche che lo distinguono da ogni altro.
Ogni elemento di una classe, cioè ogni oggetto, si dice essere istanza di quella classe, e la creazione stessa di un oggetto viene anche

detta, quindi, istanziazione dell'oggetto.
Tornando a parlare di sviluppo di software, quando programmiamo in maniera orientata agli oggetti, non dobbiamo fare altro che pensare ai concetti del dominio che stiamo affrontando, e ricavarne dunque gli oggetti e le funzionalità che devono avere e fornire, e le modalità di comunicazione fra oggetti diversi o della stessa classe. Inoltre ogni oggetto, nel mondo comune, non è un entità atomica, cioè ogni oggetto è formato da altri oggetti, il classico mouse ad esempio è formato da una pallina, da due o più tasti, magari da una rotellina, tutti oggetti che aggregati forniscono la funzionalità totale del mouse.

C++/CLI ORIENTATO AGLI OGGETTI

C++/CLI è un linguaggio orientato agli oggetti, anche se permette di programmare nella classica modalità procedurale. Quando si programma secondo il paradigma OO è necessario pensare in termini di classi e oggetti.

CLASSI E STRUTTURE

I concetti di *ref class* e *ref struct* sono nuovi di C++/CLI, ma sono fondamentalmente delle estensioni ai concetti di classe e strutture già presenti in C++.
In un linguaggio OO, creare nuovi tipi di dati vuol dire creare nuove classi. Secondo la definizione, o perlomeno una delle tante definizioni, una classe è una struttura di dati che può contenere membri dati (costanti e campi), membri funzione (metodi, proprietà, eventi, indicizzatori, operatori, costruttori di istanza, distruttori e costruttori statici) e tipi nidificati. Anche le classi *ref* e le *struct* ref contengono o sono costituite da campi e metodi. Possono esistere virtualmente infinite istanze di ogni classe, quindi ogni volta che si crea un oggetto di una data classe, in realtà si sta creando un'istanza ben di-

stinta della stessa classe, così come ogni uomo è unico anche se fa parte della stessa classe del genere umano.
Per dichiarare una *ref class* si usa la seguente sintassi:

```
ref class Poligono
{
    int numeroDiLati;
}
```

La classe *Poligono* definisce la classe delle figure geometriche con un numero arbitrario di lati, numero che sarà memorizzato nel campo *numeroDiLati* di tipo *int*. La parola chiave *ref* è la novità principale nella programmazione OOP di C++/CLI rispetto al classico C++. L'uso di *ref* per dichiarare una classe indica al compilatore che gli oggetto istanza della relativa classe *ref* sono degli oggetti di un tipi riferimento che vivono all'interno del *managed heap*. Quindi, lo sviluppatore è libero di decidere se dichiarare una classe *ref*, usufruendo dei vantaggi portati da .NET, e che si scopriranno andando avanti nei prossimi capitoli, e per esempio la garbage collecion, il fatto che ogni classe deriva implicitamente dalla superclasse *System::Object*, la possibilità di dichiarare proprietà.

Modificatori di accesso

Il blocco di una classe o di un struttura costituiscono una sorta di corpo chiuso. Dall'esterno di esso è possibile interagire solo rispettando delle regole di accesso, governate dai modificatori di accesso. Nell'esempio precedente in cui si è dichiarata la classe *Poligono*, per il campo *numeroDiLati*, non è stato specificato alcun modificatore di accesso. Una *ref class* ha come modificatore predefinito il modificatore *private*, ciò significa che i campi o i metodi della classe sono di default non accessibili dall'esterno.
Al contrario una *ref struct* ha come modificatore predefinito *public*, ciò significa che i membri di una ref struct sono di default accessibi-

li dall'esterno. Se si vuole cambiare la modalità di accesso, bisogna utilizzare appunto dei modificatori di accesso. Per poter accedere al campo *numeroDiLati* bisogna inserire il campo stesso all'interno della sezione *public*:

```
ref class Poligono
{
public:
    int numeroDiLati;
}
```

In genere il modificatore *private* si utilizza con i membri della classe che devono essere nascosti, o incapsulati nella dichiarazione della classe. Il modificatore public invece serve a definire "l'interfaccia" di utilizzo della classe, quella cioè che sarà utilizzata da un altro oggetto esterno alla classe stessa.
C++/CLI fornisce un terzo modificatore, detto *protected*. Un membro di classe definito come *protected* è un ibrido fra un membro *public* e uno *private*. Esso sarà visibile solo da classi che derivano dalla classe che definisce il membro, ma sarà nascosto al resto delle classi.

Ereditarietà di classi ref

Il concetto di ereditarietà è fondamentale nella programmazione object oriented. Ereditando da una classe madre dei metodi o dei campi, si può creare una versione specializzata della stessa classe, senza necessità di riscrivere codice o di duplicarlo. Per esempio, data la classe *Poligono*, si può definire una classe specializzata *Triangolo*, che altro on è se non un poligono speciale, con tre lati.
In C++/CLI l'unico tipo di ereditarietà consentita è quella pubblica, cioè la classe figlia può accedere ai membri pubblici i privati della classe madre. Per scrivere che *Triangolo* deriva da *Poligono* si dovrà fare così:

```
ref class Triangolo: public Poligono
{
}
```

Con la definizione precedente la classe *Triangolo*, non potrà accedere al campo *numeroDiLati*, per renderlo possibile si dovrà perlomeno trasformarlo in protected.

```
ref class Poligono
{
protected:
    int numeroDiLati;
};

ref class Triangolo:public Poligono
{
public:
    Triangolo()
    {
        numeroDiLati=3;
    }
};
```

Nell'esempio, la classe *Triangolo* possiede un costruttore pubblico, che non fa altro che inizializzare al valore 3 il campo *numeroDiLati*, ereditato dalla classe *Poligono*. Il Framework .NET permette l'ereditarietà solo da un'altra singola ref class, quindi è vietata l'ereditarietà multipla permessa nel C++ classico.

Classi *sealed*

Una classe *sealed* è una classe da cui non si può derivare una classe figlia, in tale maniera si blocca quindi la catena di ereditarietà della classe. In genere, il modificatore *sealed* si usa nei casi in cui

non si vuole che la classe sia estesa in maniera accidentale, oppure non si vuole che qualcuno possa creare una propria classe derivandola da una esistente, ad esempio per motivi commerciali. Come esempio reale, notiamo che la classe *System::String* è una classe *sealed*, per evitare che si possa creare ereditare da essa una diversa implementazione delle stringhe. Una classe *sealed* si crea semplicemente aggiungendo il modificatore sealed alla fine della dichiarazione della classe:

```
ref class Triangolo
{
	//…
}

ref class TriangoloScaleno sealed: public Triangolo
{

}

//errore di compilazione
ref class TriangoloScalenoIsoscele:TriangoloScaleno
{
}
```

La dichiarazione dell'ultima classe *TriangoloScalenoIsoscele* darà in compilazione un errore, in quanto si è tentato di derivarla dalla classe sealed *TriangoloScaleno*. Una classe astratta non può essere naturalmente anche *sealed*, non avrebbe senso d'altronde non essendo a questo punto né istanziabile né estendibile da una classe figlia.

Istanziare oggetti

Ora che si è visto come dichiarare delle classi *ref*, non resta che ve-

dere come utilizzarle, creando una istanza. Si è già parlato dell'operatore *gcnew*, che serve appunto ad istanziare una classe *ref*.
L'operatore *gcnew* è simile al classico operatore *new* di C++, con la differenza che *gcnew* crea un oggetto sullo heap CLI. Il tipo del risultato di una chiamata a *gcnew* è un handle al tipo dell'oggetto creato. Nel caso in cui si esaurisca la memoria a disposizione verrebbe lanciata una eccezione *System::OutOfMemoryException*
Volendo creare un oggetto di classe *Triangolo* basterà scrivere:

```
Triangolo^ triangolo=gcnew Triangolo();
```

L'esempio crea un handle alla classe *Triangolo*, chiamato *triangolo*. Per accedere ai membri di una classe tramite un handle si utilizza l'operatore ->. Per esempio, se la classe *Triagolo* possedesse un metodo pubblico *CalcolaArea*, si potrebbe semplicemente calcolare l'area del triangolo scrivendo:

```
triangolo->CalcolaArea();
```

È possibile anche creare una istanza sullo stack, come nel C++ classico, ed è sufficiente in questo caso scrivere:

```
Triangolo triangolo;
```

Per utilizzare i membri di un oggetto sullo stack si utilizza invece l'operatore punto ("."), quindi per il calcolo dell'area si scriverà:

```
triangolo.CalcolaArea();
```

Handle ed oggetti

È utile entrare maggiormente nel dettaglio parlando di creazione di oggetti e relativi handle. Utilizzando l'operatore *gcnew* viene allocata memoria sullo heap CLI, o managed heap (sempre che vi sia an-

cora memoria a disposizione), sullo stack invece viene creato l'handle, vale a dire la "maniglia" che permette di raggiungere e manipolare l'oggetto in memoria. Un handle è un riferimento ad un oggetto CLI, e si rappresenta con l'operatore ^ (pronuncia hat) che segue il tipo.

```
Tipo^ handle;
```

La durata della vita di un oggetto sullo heap è determinata dal processo di garbage collection di .NET. Quando l'oggetto non ha nessun handle utilizzato, ad esempio perché tutti sono usciti dal proprio scope di utilizzo (fine di un metodo, fine di un ciclo *if*, e così via), l'oggetto in memoria viene marcato come "distruggibile", e verrà cancellato appena possibile.

La Parola chiave this

C++/CLI fornisce una parola chiave particolare, *this*, che viene utilizzata all'interno di un qualsiasi metodo di una classe per riferirsi all'istanza stessa in quel momento in esecuzione. Vale a dire che *this* serve a fare in modo che l'oggetto attivo possa usare se stesso, specificandolo in modo esplicito per evitare fraintendimenti. Nella maggior parte dei casi ciò non è necessario, per esempio si prenda una classe *Veicolo*,

```
ref class Veicolo
{
    …
private: bool motoreAcceso;
public:
void AccendiMotore()
    {
        if(!motoreAcceso)
            motoreAcceso=true;
```

```
	}
}
```

Il metodo *AccendiMotore* al suo interno fa riferimento al campo *motoreAcceso*, ed è logico che esso sia un campo della stessa classe. Si può scrivere, in maniera equivalente, il metodo in questa maniera:

```
void AccendiMotore()
{
	if(!this->motoreAcceso)
		this->motoreAcceso=true;
}
```

La parola chiave *this* esplicita il fatto che si sta usando un membro dello stesso oggetto, e quando essa non è presente tale fatto è sottinteso dal compilatore. Può capitare però che l'uso di *this* si renda necessario, oltre che contribuisca a chiarire il codice, ad esempio

```
void SetDirezione(int direzione)
{
	this->direzione=direzione;
}
```

poiché il parametro ha lo stesso nome di un campo della classe, per riferirsi al campo è necessario usare *this*->direzione, mentre scrivendo all'interno del metodo solo direzione ci si riferisce alla variabile passata come parametro.

Costruttori

Un costruttore serve ad inizializzare lo stato di un oggetto, ed è un metodo speciale di una ref class che ha il nome della classe stessa ed eventualmente dei parametri. Se non fatto esplicitamente una classe ref possiede un costruttore di default senza parametri. Per

esempio l'oggetto della classe *Triangolo* è stato creato semplicemente scrivendo:

```
Triangolo^ triangolo=gcnew Triangolo;
```

Il costruttore di default non fa niente se non inizializzare ai valori default i campi e chiamare il costruttore della classe madre. Il costruttore senza parametri può naturalmente essere ridefinito in maniera da apportare le inizializzazioni che si desidera. Si tenga presente che il costruttore di default viene generato automaticamente solo se non si definisce nessun altro costruttore. Quindi se serve un costruttore senza parametri ed esistono altri costruttori, ci si ricordi di definire manualmente il costruttore senza parametri. Una speciale parte di un costruttore è la *lista di inizializzazione*. È possibile tramite questa lista fornire ed inizializzare un elenco di variabili, detti campi di classi (vedi paragrafo successivo), che necessitano di un valore prima dell'esecuzione del costruttore stesso.
La lista di inizializzazione viene scritta subito dopo il nome di un costruttore, inserendo i due punti (":") fra esso e l'elenco delle variabili da inizializzare separate dalla virgola. Il valore da assegnare ad un campo viene scritto fra parentesi tonde. Per esempio la classe *Poligono* seguente, possiede una lista di inizializzazione che imposta il valore del campo *numeroDiLati*:

```
ref class Poligono
{
protected:
	int numeroDiLati;

public:
	Poligono(int num):numeroDiLati(num)
	{
	}
```

```
};
```

Un altro utilizzo della lista di inizializzazione, forse anche più comune del precedente, è quello di invocare il costruttore della classe madre. Per esempio, definendo la classe *Triangolo* come derivata da Poligono, si può invocare il costruttore base passando il parametro 3 come numero di lati:

```
ref class Triangolo:Poligono
{
public:
    Triangolo():Poligono(3)
    {
    }
}
```

Costruttore per copia

Un particolare tipo di costruttore utilizzabile in C++/CLI è il costruttore per copia. I programmatori C++ conosceranno già tale concetto, che indica un costruttore che serve a creare una istanza di classe come copia di un'altra istanza passata come argomento al costruttore stesso:

```
Poligono(const Poligono^ p)
{
}
```

In questa maniera si può creare una copia di un *Poligono* passandolo così al costruttore per copia:

```
Poligono^ p1=gcnew Poligono(5);
Poligono^ p2=gcnew Poligono(p1);
```

Cosa succede nel caso in cui si voglia creare una copia di un *Poligono* a partire da un oggetto sullo stack e non da un handle? Per esempio:

```
Poligono p3(5);
Poligono^ p4=gcnew Poligono(p3);
```

L'errore del compilatore indicherebbe che non esiste un costruttore per copia appropriato. Una soluzione possibile è quella di ottenere l'handle all'oggetto *p3* utilizzando l'operatore unario %, quindi scrivendo:

```
Poligono^ p4=gcnew Poligono(%p3);
```

Una seconda soluzione è quella di aggiungere un altro costruttore per copia alla classe *Poligono*:

```
Poligono(const Poligono% p)
{}
```

In questo caso sarebbe possibile istanziare l'oggetto come già tentato:

```
Poligono^ p4=gcnew Poligono(p3);
```

Costruttore statico

Lo scopo di un costruttore statico è quello di inizializzare eventuali campi statici di classe (si veda più avanti). È possibile definire un costruttore statico anche in presenza di un costruttore normale (cioè non statico), e per farlo basta far precedere al costruttore la parola chiave *static*:

```
static int numPoligoni;//campo statico di classe
```

```
public:
    static Poligono()
    {
        numPoligoni=0;
    }
```

Campi di classe

I campi di una classe rappresentano dei membri che contengono i dati di un'istanza della classe stessa. In pratica sono delle variabili dichiarate all'interno del corpo di una classe. Un campo può essere di un tipo qualunque di dati, può cioè essere un oggetto o anche un campo di tipo valore. Per esempio, si supponga di voler definire una classe che rappresenti un *Veicolo* a motore:

```
ref class Veicolo
{
public:
int ruote;
    float velocita;
    int direzione;
    bool motoreAcceso;
    AutoRadio^ radio;
}
```

La classe *Veicolo* così scritta possiede uno stato interno dato dal numero di ruote del veicolo, dalla velocità, dalla direzione di movimento e dallo stato del motore, che sono dunque dati rappresentati per mezzo di tipi valore, inoltre possiede un campo che è un handle a un oggetto *AutoRadio*, cioè un tipo riferimento che per ora supponiamo di aver implementato da qualche altra parte. Come abbiamo visto prima, per istanziare un oggetto di una *ref class*, è necessario utilizzare la parola chiave *gcnew*.
Essa è seguita da un particolare metodo della classe, chiamato co-

struttore. Anche quando esso non viene esplicitamente definito, ogni classe C++/CLI ne fornisce uno standard, detto appunto costruttore di default, e che non accetta nessun parametro in ingresso. Quindi sebbene per la classe *veicolo* non abbiamo ancora pensato a un costruttore, possiamo ugualmente istanziare un oggetto:

```
Veicolo^ auto=gcnew Veicolo;
```

L'oggetto *auto* possiede i tre campi ruote, velocita e direzione, che possiamo utilizzare in lettura e in scrittura perché dichiarati *public*, quindi visibili dall'esterno. Per accedere ai membri di una classe si usa l'operatore ->; dunque, volendo ad esempio impostare a 4 il numero di ruote dell'oggetto *auto*, si scriverà:

```
auto->ruote=4;
```

I campi di una classe assumono determinati valori di default, se non vengono esplicitamente inizializzati. Questo per assicurarsi che tali campi posseggano comunque un valore dopo la creazione di un oggetto. Per esempio, un campo *bool* viene inizializzato a *false*, e un *int* sarà pari a zero.
È possibile definire dei campi di classe costanti, cioè il cui valore non può essere modificato durante il ciclo di vita dell'oggetto. Un campo costante deve essere inizializzato nel corpo di un costruttore o nella lista di inizializzazione del costruttore stesso.
Per definire un campo di classe costante si utilizza la parola chiave *const*:

```
ref class Auto
{
private:
	const int numeroRuote;
public:
```

```
    Auto(): numeroRuote(4);
}
```

A volte è molto più utile, piuttosto che assegnare un valore costante a tempo di compilazione, assegnare un valore durante l'esecuzione, come risultato di una espressione, ma non variarlo più, creare cioè un campo scrivibile solo una volta, ed a questo punto usarlo in sola lettura. Un simile campo si dichiara con la keyword initonly che precede il tipo. Un tipico esempio potrebbe essere quello di fornire ad una classe un campo *OrarioDiCreazione*, il cui valore dovrebbe essere assegnato solo una volta, appunto alla creazione dell'oggetto, e da quel momento in poi non dovrebbe essere variato.

```
class MiaClasse
{
private:
initonly DateTime OrarioDiCreazione;
    //… resto della classe
}
```

Un campo *initonly* può essere assegnato solo all'interno dei costruttori della classe, cioè al momento della creazione di un oggetto, anche come risultato di una espressione, e non solo come semplice assegnazione di un valore.

Metodi

Gli oggetti istanziati in un programma necessitano di comunicare fra di loro, ad esempio dopo aver istanziato un oggetto di classe Veicolo, come facciamo a metterlo in moto, ad accelerare e frenare? La classe *Veicolo* deve mettere a disposizione delle funzioni richiamabili dall'esterno, in modo da poter spedire alle sue istanze dei messaggi. Tali funzioni vengono in genere chiamate metodi della classe Un metodo di classe è una particolare funzione definita all'interno del

corpo di una classe. Un metodo è definito nella seguente maniera, che rappresenta la cosiddetta signature o firma del metodo:

```
tipo_ritorno NomeMetodo(lista_parametri)
{
	// corpo del metodo
}
```

Il nome del metodo è quello con cui verrà richiamato da altre parti del codice. Il tipo di ritorno determina il tipo dell'eventuale valore che deve essere ritornato dal metodo con un'istruzione *return*. Fra le parentesi che seguono il nome possono essere specificati dei parametri con rispettivi tipi, che saranno in qualche modo utilizzati nel corpo del metodo stesso. Per restituire un valore di un dato tipo viene utilizzata l'istruzione *return* seguita dal valore da restituire. A meno che il metodo non debba resituire alcun valore e quindi il tipo di ritorno venga definito come voi. In questi caso l'istruzione *return* può servire per uscire prematuramente dal metodo, senza specificare alcun valore di ritorno. Supponendo di avere allora la classe *Veicolo*, e di voler definire il metodo di accelerazione del veicolo stesso, si potrebbe definire una funzione, nella parte di classe *public*, in maniera che il metodo sia accessibile dall'esterno, chiamandolo per esempio *Accelera* e passando un parametro *int* che indica la velocità da raggiungere:

```
ref classe Veicolo
{
public:
	void Accelera(int vel)
	{
		if(vel<velocitaMax)
			velocita=vel;
		else velocita=velocitaMax;
```

```
}
}
```

Metodi virtuali

Nel paradigma di programmazione ad oggetti, i metodi virtuali costituiscono un punto fondamentale. I metodi virtuali mettono in pratica quello che viene chiamato polimorfismo. Il polimorfismo è una caratteristica della programmazione orientata agli oggetti, strettamente legata all'ereditarietà. Il polimorfismo indica la capacità degli oggetti di assumere comportamenti diversi a seconda della classe di appartenenza. Ciò avviene effettuando l'overriding dei metodi ereditati nelle classi figlie, ma prevedendo già nell'implementazione della classe madre quali dei suoi metodi saranno sottoposti a sovraccarico nelle classi derivate. Ciò si ottiene in due possibili maniere. La prima è quella di dichiarare il metodo della classe base con la parola chiave virtual, ed il metodo relativo delle classi figlie con la parola chiave *override*.
Si supponga di derivare dalla classe ref Veicolo, una classe figlia Moto. Nella classe Veicolo poi si supponga di implementare il metodo *Accelera* come *virtual*.
La classe *Moto*, può fornire una propria implementazione del metodo di accelerazione, in quanto una Moto accelera in maniera diversa da una automobile. Il polimorfismo si manifesta quando il metodo Accelera viene invocato attraverso un handle dalla classe madre *Veicolo*. Per esempio:

```
Veicolo^ veicolo=gcnew Moto();
veicolo->Accelera();
```

La chiamata al metodo *Accelera* eseguira la versione *Accelera* implementata all'interno della classe *Moto*, se il metodo è stato dichiarato *virtual*. Tecnicamente accade che ciò che determina quale metodo invocare è il tipo dell'oggetto sul quale si chiama il metodo, in

questo caso *Moto*. Ecco come utilizzare la parola chiave *virtual*, inserendola prima della firma del metodo stesso:

```
virtual void Accelera()
{}
```

Se un metodo è *virtual* in una classe base, lo sarà implicitamente anche nelle classi figlie.

Override dei metodi

Quando si implementa una nuova versione di un metodo virtual in una classe figlia si parla di override.
L'override di un metodo può essere implicita o esplicita. Con override implicido si intende la creazione di un metodo nella classe figlia, con lo stesso nome e gli stessi parametri, continuando ad usare la parola chiave virtual, e facendo seguire al nome del metodo la parola chiave *override*:

```
virtual void Accelera() override
{
}
```

È possibile anche nascondere un metodo *virtual*, creandone uno nuovo con lo stesso nome e firma, ed utilizzando la parola chiave *new* al posto di *override*. In questa maniera si interrompe la propagazione della catena virtuale lungo le generazioni di classi figlie.

```
void Accelera() new
{
}
```

L'ultima possibilità non è molto obejct oriented, ma per qualche motivo potrebbe servire. È poi possibile il cosiddetto override esplicito

di un metodo virtuale, esso consiste nell'implementare in una classe figlia un metodo con un nome diverso da quello virtuale della classe madre, ma indicando il nome completo del metodo che si vuole sovrascrivere:

```
ref class Veicolo
{
…
    virtual void Accelera()
    {
        velocita+=2;
    }
};

ref class Moto: public Veicolo
{
public:
    Moto():Veicolo(2)
    {
    }

    virtual void AcceleraMoto() = Veicolo::Accelera
    {
        velocita++;
    }
};
```

Il metodo *AcceleraMoto* della classe *Moto* effettua l'override del metodo *Accelera* di *Veicolo*, come viene esplicitamente indicato nella firma stessa del nuovo metodo. È necessario che sia il metodo *Accelera* della classe madre, che il nuovo *AcceleraMoto* siano segnati come *virtual*.

Metodi virtuali puri

In certi casi è utile e desiderabile fare in modo che una classe figlia implementi necessariamente un override di un metodo virtuale. Per fare ciò è necessario che il metodo della classe madrea sia un metodo virtuale puro. Una classe che contiene un metodo virtuale puro è una classe che non può essere istanziata, cioè una classe astratta, ed un metodo virtuale puro è quoindi una specie di segnaposto, che indica la firma di un metodo, ma non fornisce alcuna implementazione del suo corpo.
La sintassi per dichiarare un metodo virtuale puro prevece l'aggiunta di un *=0* subito dopo la firma di un metodo virtuale:

```
virtual void Accelera() = 0;
```

Inserendo tale dichiarazione nella classe *Veicolo*, tutte le sue figlie saranno obbligae ad implementare un metodo *Accelera*.

Overloading dei metodi

L'overloading di un metodo consiste nel sovraccaricare un metodo con diverse possibili firme, fornendo diverse possibilità per invocarlo, in termini di numero e/o tipi di parametri da passare ad esso. Per esempio è possibile creare un metodo *Accelera* che prende in ingresso un int raprresentante la velocità finale da raggiungere, ed esso sarebbe un overload del precedente metodo *Accelera*.

```
void Accelera() …{};
void Accelera(int velocita) …{};
```

Invocare un overload non è niente di speciale, basta passare i parametri giusti per eseguire il metodo con la firma coincidente.

Overloading degli operatori

L'overloading degli operatori è una delle carateristiche più impor-

tanti che si impara nello studio del linguaggio C++, e dunque lo è anche in C++/CLI. L'overloading di un operatore permette di usare l'operatore standard sovraccaricato con oggetti di una classe *ref*, per esempio per unire due stringhe utilizzando un semplice + piuttosto che scrivendo un metodo *Concatena*. In C++/CLI la sintassi per l'overloading degli operatori è leggermente differente rispetto a quella di C++ standard. Non tutti gli operatori possono essere sovraccaricati, e fra quelli mancanti, citiamo le parentesi [], o l'operatore *gcnew*.

Operatori unari

Gli operatori unari sono quelli che agiscono su un solo operando. In genere l'operando stesso non viene modificato dall'operatore, ad esclusione di quelli di incremento e decremento. Con gli operatori unari che agiscono su tipi riferimento si può fare in modo che l'operando venga invece anch'esso modificato.
Supponendo di avere una classe *MiaClasse*, e che si voglia definire un overloading dell'operatore unario +, ecco come fare:

```
static MiaClasse^ operator +(const MiaClasse ^operando)
{
MiaClasse^ ret = gcnew MiaClasse();
//esegue qualche operazione
return ret;//restituisce l'oggetto corrispondente al risultato
+MiaClasse
}
```

La presenza del const prima del parametro, garantisce che l'operando rimanga immutato. Se si vuole proprio modificare in qualche modo l'operando stesso basta togliere il *const* ed agire sulla variabile operando:

```
static MiaClasse^ operator +( MiaClasse ^operando)
```

```
{
//esegue qualche operazione su operando
return operando;//restituisce l'oggetto corrispondente al risultato
                                                          +MiaClasse
}
```

Operatori binari

Per effettuare l'overloading di un operatore binario, cioè che agisce su due operandi, si può anche definire il tipo di ritorno, che non è detto sia il medesimo degli operandi, e passare due operandi al metodo che definisce il comportamento dell'operatore. Il seguente è un esempio di ridefinizione dell'operatore * per la classe *String*:

```
static int operator *(const String ^lhs, const String ^rhs)
{
return lhs->Length *  rhs->Length;
}
```

Moltiplicando due stringhe il risultato sarebbe il prodotto della lunghezza delle stringhe stesse. Non ha molto senso, ma serve come esempio.

Parametri opzionali

Un metodo con un numero variabile o opzionale di argomenti ha come suo ultimo parametro un array speciale, chiamato *parameter array*. Può esserci un solo array di parametri per ogni metodo, e deve essere necessariamente l'ultimo. Un array di parametri viene specificato utilizzando un prefisso di tre puntini. Per esempio un metodo che prende come argomento un array di parametri può essere definito così:

```
void StampaArgomenti(String^ primo,...array<String^>^ args)
{
```

```
    Console::WriteLine(primo);
    for each(String^ str in args)
        Console::WriteLine(str);
}
```

Il primo argomento è una stringa, mentre il secondo è un array opzionale. Ciò significa che il metodo *StampaArgomenti* può essere invocato passando come parametro una sola stringa, oppure due, tre e così via. Per esempio, tutte le seguenti chiamate sono lecite:

```
StampaArgomenti("a"); // l'array di parametri è vuoto
StampaArgomenti("a","b"); // l'array di parametri ha 1 solo elemento
```

Proprietà

Una proprietà è un particolare membro di classe che si comporta come se fosse un campo. I programmatori C# saranno già abituati a tale concetto, anzi lo troveranno normale. Data la mancanza di un tale costrutto in C++, esso invece costituisce una importante novità di C++/CLI.

È possibile definire in C++/CLI due tipi di prorietà differenti. Il primo è detto proprietà scalare, ed esso permette di accedere ad un membro come se fosse un campo, la differenza è che un campo rappresenta una locazione di memorizzazione, mentre una proprietà permette una sorta di accesso controllato, infatti è possibile eseguire altro codice quando la proprietà stessa deve essere scritta o letta. Degli esempi di proprietà possono essere la lunghezza di una stringa, o il numero di elementi di un vetore. Una proprietà viene definia utilizzando la parola chiave *property* che precede il tipo ed il nome della proprietà stessa. Subito dopo seguono i due rami di accesso in lettura o scrittura, non necessariamente entrambi presenti. Per esempio una proprietà potrebbe essere di sola lettura, e quindi si implementerà solo il ramo get, oppure di sola scrittura, implementando dunque il ramo set. Riprendendo l'esempio del veicolo, è possibile

definire una proprietà pubblica che restituisce o imposta il numero di ruote, memorizzato in un campo privato, e quindi altrimenti irraggiungibile.

```
property float Velocita
{
	float get()
	{
		return velocita;
	}
	void set(float value)
	{
		velocita=value;
	}
}
```

La funzione *get* viene invocata quando la proprietà *Velocita* deve essere letta. In questo caso essa restituisce semplicemente il valore del campo privato *velocita*.

```
Veicolo^ v=gcnew Veicolo;
int  velocita=v->Velocita;
```

Il ramo di accesso *set* invece viene invocato quando la proprietà viene scritta, cioè quando si assegna ad essa un valore:

```
v->Velocita=10;
```

L'uso delle proprietà è molto semplice e diretto, ed inoltre permette una programmazione molto intuitiva ed allo stesso tempo più sicura, potendo inserire dei controlli sia durante la lettura che durante la scrittura di una proprietà. In C++/CLI è possibile utilizzare una scrittura rapida delle proprietà per accedere in lettura o scrittura ad un cam-

po di classe. In questo modo si dirà che so tratta di una dichiarazione di proprietà triviale. Per esempio:

```
property String^ Targa;
```

il compilatore provvede automaticamente a generare le implementazioni di default per i due rami get e set e si potrà scrivere la proprietà così:

```
veicolo->Targa="AA123BB";
```

oppure leggerla così:

```
String^ targa=veicolo->Targa;
```

Il secondo tipo di proprietà permesse da C++/CLI è quello delle proprietà indicizzate o indicizzatori di classe. Una proprietà indicizzata permette di accedere ad una classe come se fosse un array, utilizzando cioè l'operatore []. Come esempio, si consideri il caso in cui si abbia una classe *Treno*, formata da diversi vagoni, e si voglia dare la possibilità di accedere ai singoli vagoni utilizzando l'accesso tipo array.
La definizione di una proprietà indicizzata è simile all'appena vista definizione di proprietà scalari, con la differenza principale che una proprietà di indicizzazione non ha un nome, ed include il parametro da utilizzare come indice:

```
ref class Treno
{
    property Vagone^ default[int]
{
Object^ get(int index)
{
```

```
return GetVagone(index);
}
}
}
```

Con tale definizione si potrebbe ottenere il primo vagone del treno scrivendo:

```
Treno^ treno=gcnew Treno;
…
Vagone^ vagone1=treno[0];
```

Classi innestate

È possibile definire delle classi innestandole all'interno di un'altra dichiarazione di classe. Le classi innestate permettono di creare una sorta di relazione del tipo "Contiene". Naturalmente la stessa cosa può essere ottenuta definendo la classe contenuta in maniera totalmente separata, e poi utilizzandola all'interno della classe contenitrice. Ciò che permette però di fare il concetto di classe innestata è che quest'ultima sarò utilizzabile e visibile appunto solo dalla classe che la contiene, nascondendola complemtamente al resto del mondo.

```
ref class Contenitore
{
public:
ref class ClasseInterna
{
public:
int publicMember;
};
private:
ClasseInterna^ obj; // handle alla classs innestata ClasseInterna
```

```
}
```

CLASSI ASTRATTE

Una classe astratta è perfettamente identica ad una ref class normale, quindi può avere campi, metodi, proprietà e così via, l'unica differenza è, come dice il nome, è che essa non può essere mai istanziata. Qual è dunque l'utilità di un tale tipo? Da essa possono essere derivate altre classi, non astratte e quindi istanziabili, che condividono i membri *protected* o *public* della classe madre. Una classe astratta può anche possedere un costruttore, il cui scopo è quello classico di inizializzare lo stato dell'oggetto. L'unico posto dove poter invocare il costruttore della classe astratta, è la lista di inizializzazione delle classi figlie, e quindi è consigliabile dichiarare il costruttore stesso con il modificatore *protected*. Una classe contenente almeno un metodo virtuale puro è una definizione di classe astratta per natura. In tale caso, infatti, la keyword *abstract* introdotta da C++/CLI è opzionale e serve ad esplicitare la natura della classe agli occhi degli sviluppatori meno esperti o attenti. La parola chiave *abstract* deve essere aggiunta dopo la dichiarazione della classe stessa:

```
ref class AbstractVeicolo abstract
{
protected:
 int velocita;
public:
virtual void Accelera() = 0; // unimplemented method
void StampaVelocita()
{
Console::WriteLine("velocita attuale : {0}",velocita);
}
};
```

La classe *AbsractVeicolo* è una classe che possiede il solito campo int velocita ed un metodo virtuale puro da implementare nelle classi figlie. A questo punto si può derivare da *AbstractVeicolo* la classe *Automobile*:

```
ref class Automobile: AbstractVeicolo
{
public:
	virtual void Accelera() override sealed
{
velocita+=2;
}
}
```

Istanziando ora la classe *Automobile* e provando ad invocare i metodi:

```
Automobile^ automobile=gcnew Automobile();
automobile->Accelera();
automobile->StampaVelocita();
```

Verrebbe eseguito il metodo *Accelera* della classe *Automobile*, override dell'omonimo metodo virtuale puro della classe madre, e poi utilizzando il metodo *StampaVelocita* ereditato, verrebbe stampato il messaggio:

```
velocità attuale : 2
```

INTERFACCE

Un'interfaccia è un contratto applicabile ad una classe. Una classe che si impegna a rispettare tale contratto, cioè ad implementare un'interfaccia, promette che implementerà tutti i metodi, le proprietà, gli

eventi, e gli indicizzatori, esposti dall'interfaccia. Il concetto è molto simile a quello di classe astratta, ma nel caso di una interfaccia non è possibile fornire alcuna implementazione dei membri di cui sopra, né essa può dunque contenere dei campi, ed inoltre il tipo di relazione di una classe derivante da una classe astratta è sempre una relazione di generalizzazione, nel caso invece di una classe che implementa un'interfaccia, la relazione è appunto di implementazione. Ad esempio un *Triciclo* potrebbe derivare dalla classe astratta *VeicoloAPedali*, e potrebbe invece implementare un'interfaccia che espone i metodi per guidarlo, interfaccia implementabile anche da altri tipi, ad esempio dalla classe *Shuttle* che non è un *VeicoloAPedali*, ma deve essere guidabile, quindi è naturale pensare ad una interfaccia *IGuidabile*.
In C++/CLI, una interfaccia viene dichiarata utilizzando le due keyword interface class. Per creare un'interfaccia, è sufficiente in parole povere, far precedere alla parola chiave class, la parola chiave *interface* al posto di *ref*, e nel corpo dell'interfaccia inserire solo dei metodi virtuali puri e pubblici.
Le linee guida .NET suggeriscono di chiamare le interfacce con un nome del tipo *INomeInterfaccia*, un esempio reale è l'interfaccia *IComparable* implementata dalle classi per cui si vuole fornire un metodo di comparazione e ordinamento.
Tutti i membri esposti da un'interfaccia servono in pratica solo da contratto o segnaposto per quelle che saranno poi le loro implementazioni. Supponiamo di voler definire una interfaccia *IAtleta*. Un atleta deve poter correre e fermarsi. Quindi l'interfaccia potrebbe essere questa:

```
interface class IAtleta
{
public:
    virtual void Corre() = 0;
    void Salta();
```

```
}
```

Si noti che è possibile tralasciare l'uso della parola chiave *virtual* o del suffisso *=0*, perché trovandosi in un'interfaccia, si assume che tutti i metodi sono virtuali puri. Nell'esempio precedente, il secondo metodo dell'interfaccia non ha né *virtual* né *=0*. Le interfacce devono essere implementate utilizzando l'ereditarietà pubblica, e se la *keyword public* viene omessa, si assume che l'ereditarietà sia ancora *public*:

```
interface class Interfaccia
{
};

ref class Classe1 : private Interfaccia {}; //errore di compilazione C3141
ref class Classe2 : protected Interfaccia {}; //errore di compilazione C3141
ref class Classe3 : Interfaccia {}; //public assumed
```

Provando ora a implementare l'interfaccia *IAtleta* prima definita, è necessario fornire un'implementazione *public* dei due metodi che essa ha definito:

```
ref class Calciatore: public IAtleta
{
private:
	String^ nome, ^cognome;
public:
	virtual void Corre()
	{
		Console::WriteLine("Sto correndo");
	};

	virtual void Salta()
	{
```

```
            Console::WriteLine("Sto saltando");
      };
};
```

Come per le classi astratte, non si può istanziare una interfaccia. Ma è possibile istanziare naturalmente una ref class che implementa un'interfaccia e quindi accedere ai metodi o alle proprietà della ref class stessa utilizzando un handle all'interfaccia:

```
IAtleta^ atleta=gcnew Calciatore();
atleta->Corre();
```

Ciò consente di scrivere delle classi generiche che operano su un'interfaccia.
Una classe può anche implementare più interfacce contemporaneamente, in questo modo un oggetto è una sola cosa, ma si comporta in modi diversi, in maniera polimorfica.
Ad esempio se volessimo confrontare due calciatori, potremmo implementare contemporaneamente alla *IAtleta*, l'interfaccia *IComparable* che espone il seguende metodo *CompareTo()*:

```
virtual int CompareTo(Object^ obj);
```

Il valore di ritorno può essere minore,uguale o maggiore di zero, ed è utilizzato per dire se l'istanza è minore, uguale o maggiore di *obj*, secondo un certo parametro di ordinamento.
Quindi possiamo confrontare due *Calciatori*, confrontando nome, cognome (in un mondo reale il confronto forse non sarebbe sufficiente):

```
ref class Calciatore: public IAtleta, IComparable
{
private:
```

```
	String^ nome, ^cognome;
public:
	virtual void Corre()
	{
		Console::WriteLine("Sto correndo");
	};

	virtual void Salta()
	{
		Console::WriteLine("Sto saltando");
	};

	virtual int CompareTo(Object ^obj)
	{
		if ( dynamic_cast<Calciatore^>(obj) != nullptr)
		{
			Calciatore^ c=dynamic_cast<Calciatore^>(obj);
			if(nome==c->nome && cognome==c->cognome)
				return 0;
		}
		return -1;
	}
};
```

C++/CLI come detto consente l'ereditarietà singola delle classi, ma anche l'ereditarietà multipla delle interfacce, vale a dire che un'interfaccia può derivare da più interfacce contemporaneamente, ad esempio supponiamo di voler scrivere un'interfaccia *IAtletaUniversale* e che definisce il comportamento di un'atleta con più capacità atletiche, partendo da interfacce che definiscono le singole capacità:

```
interface class INuotatore
{
```

```
public:
    virtual void Nuota() = 0;
}

interface class ISciatore
{
    void Scia();
}

interface class ITennista
{
    void Dritto();
    void Rovescio();
}

interface IAtletaUniversale: INuotatore, ISciatore,ITennista,IAtleta
{
    void Mangia();
    void Dormi();
}
```


Una classe che implementi l'interfaccia *IAtletaUniversale*, deve non solo fornire un corpo per i metodi *Mangia* e *Dormi*, ma anche tutti quelli delle interfacce da cui *IAtletaUniversale*, deriva:

```
ref class AtletaCompleto:IAtletaUniversale
{
public:
    virtual void Mangia()
    {
        Console::WriteLine("Mangio");
    }
    virtual void Dormi()
```

```
{
        Console::WriteLine("Dormo");
}

virtual void Nuota()
{
        Console::WriteLine("Nuoto");
}
virtual void Scia()
{
        Console::WriteLine("Scio");
}
virtual void Dritto()
{
        Console::WriteLine("Colpisco con il dritto");
}
virtual void Rovescio()
{
        Console::WriteLine("Colpisco di rovescio");
}
virtual void Corre()
{
        Console::WriteLine("Corro");
}
virtual void Salta()
{
        Console::WriteLine("Salto");
}

virtual void Esulta(void)
{
        Console::WriteLine("Siiiiiii");
}
```

```
};
```

Se una classe deriva da una classe base ed implementa qualche interfaccia, è necessario che nella dichiarazione venga prima la classe, seguita dall'elenco delle interfacce. Potrebbe naturalmente accadere che più interfacce espongano un metodo con la stessa firma, ad esempio supponiamo che le interfacce *ITennista* e *ISciatore* espongano un metodo *Esulta*, come implementarlo e quante volte nella classe concreta *AtletaCompleto*? Il modo più semplice è fornire un solo metodo con lo stesso nome:

```
virtual void Esulta()
{
	Console::WriteLine("Siiiiiiii");
}
```

In questo caso, sia con un *ISciatore* che con un *ITennista*, verrebbe invocato l'unico *Esulta()* disponibile:

```
ISciatore^ sciatore=gcnew AtletaCompleto;
ITennista^ tennista=gcnew AtletaCompleto;
sciatore->Esulta();
tennista->Esulta();
```

Override esplicito

L'esempio precedente stampa due volte esattamente la stessa riga. Ma è possibile anche implementare due versioni diverse di *Esulta*, in questo caso è necessario risolvere l'ambiguità nella classe che implementa le due interfacce, indicando esplicitamente quale metodo di quale interfaccia si sta implementando:

```
ref class AtletaCompleto:IAtletaUniversale
{
```

```
    …
    virtual void EsultaTennista() = ITennista::Esulta
    {
        Console::WriteLine("Sono il miglior tennista");
    }

    virtual void EsultaSciatore() = ISciatore::Esulta
    {
        Console::WriteLine("Scio come Thoeni");
    }
}
```

Si noti che l'implementazione di due metodi in una stessa classe non può avere la stessa firma, quindi è necessario cambiare nome, ed indicare l’interfaccia ed il metodo da implementare subito dopo, con la sintassi:

```
virtual void NomeMetodo() = Interfaccia1::MetodoDaImpl,
                                        Interfaccia2::MetodoDaImpl,…
{}
```

Implementando così entrambi i metodi *Esulta* di *ISciatore* ed *ITennista*, come si fa a invocare quello giusto? Scrivendo semplicemente come nella riga seguente:

```
atleta->Esulta(); //errore C2668 chiamata ambigua ad un metodo
```

Non si ha modo di capire quale metodo *Esulta* si intenda eseguire. Allora è necessario effettuare un cast al tipo dell’interfaccia, in modo da risolvere il tipo che si vuole effettivamente utilizzare, per esempio:

```
((ITennista^)atleta2)->Esulta();
```

```
((ISciatore^)atleta2)->Esulta();
```

Si sarà notato che la sintassi generale per l'override esplicito dei metodi di un'interfaccia consente di specificare più metodi sulla stessa riga, separati dalla virgola. La cosa si dimostra utile quando ci sono da implementare diversi metodi di interfacce, alcuni dei quali non si vogliono meglio specificare, o si vuole rimandare la loro implementazione reale. Si supponga che un'interfaccia contenga dieci metodi da implementare, e per il momento ne servano un paio. Per gli altri, obbligatoriamente da implementare in quanto il contratto dell'interfaccia lo stabilisce e lo impone, potremmo utilzzare un solo metodo segnaposto:

```
interface class InterfacciaCon10Metodi
{
    void M1();
    void M2();
    void M3();
    void M4();
    void M5();
    void M6();
    void M7();
    void M8();
    void M9();
    void M10();
};

ref class ClassePigra: InterfacciaCon10Metodi
{
public:
    virtual void MetodoImpl() =
            InterfacciaCon10Metodi::M9,InterfacciaCon10Metodi::M10
    {
```

```
		Console::WriteLine("M9 e M10");
	};

	virtual void MetodoNonImpl() = InterfacciaCon10Metodi::M1,
		InterfacciaCon10Metodi::M2,
		InterfacciaCon10Metodi::M3,
		InterfacciaCon10Metodi::M4,
		InterfacciaCon10Metodi::M5,
		InterfacciaCon10Metodi::M6,
		InterfacciaCon10Metodi::M7,
		InterfacciaCon10Metodi::M8
	{
		Console::WriteLine("Metodo non implementato");
	}
};
```

I due metodi *M9* e *M10* sono implementati dall'unico *MetodoImpl*, i restanti otto dal metodo *MetodoNonImpl*.

CONVERSIONI DI TIPO

Parlando dei tipi predefiniti del linguaggio C++/CLI si è visto anche come effettuare una conversione da un tipo all'altro. Ora lo stesso argomento verrà affrontato in relazione alla conversione fra classi e/o strutture. C++/CLI fornisce tre operatori per effettuare un cast fra classi o struct: *static_cast*, *dynamic_cast*, e *safe_cast*. Ognuno di essi effettua un tentativo di conversione, e naturalmente esso non è deto sia consentito o lecito, in quanto potrebbe non esserci alcun modo di convertire un tipo verso un altro, e probabilmente in questo caso non avrebbe senso, come cercare di moltiplicare un numero per una casa, in genere non si può fare. In generale perché un tipo di una data classe sia convertibile in un altro, esso deve essere perlomeno di una classe che deriva dalla prima. Per esempio, se un ogget-

to fosse di tipo *Moto*, classe derivata dalla classe *Veicolo*, allora l'oggetto Moto è convertibile in un *Veicolo*. Ciò implica, che essendo ogni tipo derivato in .NET dalla superclasse *System::Object*, ogni oggetto è convertibile perlomeno in un oggetto *Object*. Non è possibile il contrario, un *Veicolo* non è detto che sia una *Moto*, può essere anche una *Automobile*, e quindi non sempre esso sarà convertibile in *Moto*.

L'operatore dynamic_cast

L'operatore *dynamic_cast* verifica che la conversione di tipo sia valida, ed in caso affermativo effettua il cast richiesto. Al contrario se la conversione è illecita, esso restituisce il valore *nullptr*. La sintassi di utilizzo dell'operatore è la seguente:

```
dynamic_cast<TipoDestinazione^>(varOrigine);
```

In C# esiste un operatore che serve a determinare se un oggetto è di una data classe, l'operatore *is*. In C++ l'operazione di verifica può essere implementata in maniera un po' più complessa, utilizzando l'operatore *dynamic_cast*:

```
//C#
if(obj is Veicolo)
{
// obj è un Veicolo
}
//C++/CLI
if ( dynamic_cast<Veicolo^>(obj) !=nullptr)
{
// obj è un Veicolo
}
```

In C# esiste un altro operatore, *as*, che permette di effettuare una

conversione fra tipi compatibili, ed in caso di non riuscita, restituisce un oggetto nullo. In C++/CLI tale comportamente è implementabile ancora con l'operatore *dynamic_cast*:

```
//C#
Veicolo v=obj as Veicolo; //se obj non è convertibile in Veicolo, v diventa null
//C++/CLI
Veicolo^ v= dynamic_cast<Veicolo^>(obj); //se obj non è convertibile in
                                                    Veicolo, v diventa nullptr
```

L'operatore static_cast

L'operatore *static_cast* è il più efficiente in termini di prestazioni, ma è anche il più pericoloso fra i tre operatori di conversione. Esso presuppone che lo sviluppatore sappia cosa stia facendo, in quanto non viene effettuata alcuna verifica che l'operazione di cast sia possibile. La sintassi dell'operatore è la medesima già vista per *dynamic_cast*:

```
static_cast<TipoDest>(objOrig);
```

L'operatore safe_cast

L'ultimo degli operatori di conversione è *safe_cast*. Esso, per chi già ha avuto modo di programmare in altri linguaggi .NET come C#, sembrerà il più naturale, nel senso che il suo comportamento è quello che si incontra nelle conversioni di tipo di ogni altro linguaggio.
In effetti, C++/CLI utilizza internamente tale operatore anche quando si effettua un cast fra variabili di tipo numerico:

```
int a=(int)2.2;
```

è interpretato dal compilatore come:

```
int a=safe_cast<int>(2.2);
```

Quindi in generale basta utilizzare le parentesi senza necessità di scrivere con la sintassi prevista da *safe_cast*. La sintassi dell'operatore safe_cast è praticamente identica a quella degli altri, ed il comportamento è simile. Per esempio come l'operatore *dynamic_cast* esso tenta di effettuare una conversione, e se non vi riesce, piuttosto che restituire un *nullptr*, esso lancia una eccezione di tipo *Sstem::InvalidCastException*. Si vedrà fra poco cos'è un'eccezione, per chi non conoscesse il concetto, basti pensare ad un errore generato a runtime.

CONCETTI AVANZATI

Dopo aver visto le basi della sintassi di C++/CLI, i vari operatori e le istruzioni per controllare il flusso di un programma, ed aver affrontato i concetti della programmazione oop come intesi da C++/CLI, si è pronti per affrontare argomenti leggermente più avanzati, ma anch'essi assolutamente da conoscere per programmare applicazioni di utilità ed utilizzo reale.

DELEGATE ED EVENTI

I delegate possono essere considerati l'equivalenti in .NET e quindi in C++/CLI dei puntatori a funzioni di C++. Un delegate può nascondere uno o più metodi, che abbiano una firma coincidente a quella dichiarata dal delegate stessi, ed invocare tali metodi, con l'invocazione del delegate, come se ach'esso fosse un metodo. Invocando dunque un delegate, si invoca in maniera indiretta il metodo o i metodi che il delegate maschera al suo interno, anche senza sapere quale metodo il delegate contenga. La differenza fondamentale fra delegate e puntatori a funzione è che innanzitutto il meccanismo dei delegate è type-safe, è object oriented, ed è sicuro, perché come si vedrà meglio fra qualche riga, i delegate non sono altro che delle istanze di normalissime classi, con dei metodi implementati per invocare altri metodi. In parole povere, è possibile dunque usare un metodo come se fosse un oggetto qualunque, e dunque passarlo come parametro ad un altro metodo. I delegate sono fortemente associati ad un altro concetto: quello di *evento*. Anche gli eventi sono degli oggetti .NET. Essi consentono in parole povere di implementare dei meccanismi di notifica.
Un evento viene dichiarato come un handle ad un tipo di delegate. Se in una data classe può verificarsi un particolare evento, è possibile da una o più classi diverse, mettersi in ascolto del verificarsi di

tale evento, e quindi intraprendere una particolare azione. Per poter gestire tale evento la classe che si mette in attesa di esso, deve implementare un metodo, chiamato gestore di evento, la cui firma combaci con quella definita dal delegate a cui l'evento è stato dichiarato come handle.
Per capire meglio tali concetti, forse duri da comprendere a parole, si vedrà qualche esempio, ricordando che tali meccanismi sono pesantemente utilizzati dalle librerie standard di .NET, per esempio si pensi alle Windows Forms, ed ai vari eventi che accadono quando si clicca su un pulsante, quando si muove il mouse su un controllo, quando si chiude una finestra e così via.

Delegate

Un *delegate* è una ref class che accetta e poi invoca uno o più metodi con la stessa firma da esso definita, e che si trovano in una classe, oppure che siano funzioni globali. C++/CLI supporta la creazione di delegate del tipo *System::MulticastDelegate*, cioè di un delegate che può accettare ed invocare un insieme di metodi, detta catena, in sequenza. È possibile naturalmente che tale catena sia composta da un solo metodo. Per rendere più semplice e comprensibile l'argomento, si vedrà un semplice esempio pratico, suppondendo di voler realizzare un'applicazione bancaria, con una classe *Banca*, all'interno della quale sono contenuti i conti correnti dei clienti.

Dichiarazione di un delegate

La dichiarazione di un delegate è perfettamente analoga alla dichiarazione di un metodo. Tale dichiarazione specifica la firma del metodo, cioè i parametri di ingresso e il tipo di ritorno, che il delegate stesso maschera. Tale metodo potrà poi essere sia un metodo statico che un metodo di istanza. Nel primo caso il delegate incapsula unicamente il metodo che dovrà invocare per conto di qualcun altro, nel secondo caso esso incapsula l'istanza ed un metodo di essa che dovrà invocare.

In generale la dichiarazione di un delegate avviene dunque con la seguente sintassi:

```
delegate tipo_ritorno NomeDelegate([parametri]);
```

Dunque la dichiarazione è analoga a quella di un metodo qualunque con l'aggiunta della parola chiave delegate, e senza naturalmente fornirne il corpo, ma in realtà la dichiarazione nasconde quella di una classe derivata dalla classe *System::MulticastDelegate*. Per convenzione, si usa spesso chiamare i delegate facendo terminare il loro nome per "Delegate". Un metodo ed un delegato saranno compatibili, se essi hanno lo stesso tipo di ritorno e gli stessi parametri, sia come numero, sia come tipo, che come ordine di apparizione. Riprendendo l'esempio, la seguente dichiarazione di delegate può essere utilizzata per riferirsi a metodi che prendono in ingresso un parametro di tipo *String* e non restituiscono un valore di ritorno:

```
delegate void SaldoDelegate(ContoCorrente c);
```

Lo scopo è quello di fornire un metodo per la visualizzazione del saldo di un numero di conto corrente, tale saldo potrà essere fornito con un messaggio a video, con una stampa su file, con l'invio di un e-mail, e in mille altri modi. La classe *ContoCorrente* ad esempio potrebbe avere due metodi diversi che rispettano la stessa firma specificata dal delegate, e per completezza uno dei due lo implementeremo statico:

```
ref class ContoCorrente
{
private:
	int numeroConto;
	float saldoAttuale;
public:
```

```
ContoCorrente(int numero):numeroConto(numero)
{}
void Preleva(float f)
{
    saldoAttuale-=f;
}
void Versa(float f)
{
    saldoAttuale+=f;
}

void VisualizzaSaldo(ContoCorrente^ conto)
{
    Console::WriteLine("saldo del conto {0}: {1}",conto-
                        >numeroConto,conto->saldoAttuale);
}

static void SalvaSaldoSuFile(ContoCorrente^ conto)
{
    StreamWriter^ sw=nullptr;
    try
    {
        sw=File::CreateText("c:\\temp\\saldo_"+conto-
                            >numeroConto+".txt");
        sw->WriteLine("saldo del conto {0}: {1}", conto-
                        >numeroConto,conto->saldoAttuale);
    }
    catch(IOException^ ioe)
    {
        Console::WriteLine(ioe->Message);
    }
    finally
    {
```

```
                if(sw!=nullptr)
                        sw->Close();
            }
    }

};
```

Il metodo *VisualizzaSaldo* mostra sulla console il saldo del conto, mentre il metodo *SalvaSaldoSuFile* lo scriverà su un file di testo.

Istanziazione e invocazione di un delegate

Un delegate è dietro le quinte una classe derivata dalla classe *System::MulticastDelegate*, dunque l'istanziazione avviene per mezzo della parola chiave gcnew, come per qualsiasi altro oggetto, e passando come argomento un metodo che rispetti la firma definita dal delegato stesso. Ad esempio per creare un'istanza del precedente delegate *SaldoDelegate*, delegato del metodo *VisualizzaSaldo*, è sufficiente passare al costruttore l'istanza della classe e l'indirizzo del metodo:

```
ContoCorrente ^conto = gcnew ContoCorrente();
SaldoDelegate^  saldoVideo= gcnew SaldoDelegate(conto,
                                        &ContoCorrente::VisualizzaSaldo);
//se il delegate è definito all'intero della classe
ContoCorrente::SaldoDelegate^ saldo2=gcnew
   ContoCorrente::SaldoDelegate(conto, &ContoCorrente::VisualizzaSaldo);
```

Mentre la sintassi per utilizzare il secondo metodo che è statico è la seguente:

```
//metodo Statico
SaldoDelegate ^saldoFile = gcnew
```

```
SayDelegate(&ContoCorrente::SalvaSaldoSuFile);
```

Aggiungiamo allora alla classe *ContoCorrente* un metodo generico *ElaboraSaldo* che invocherà il delegate passato come argomento:

```
void ElaboraSaldo(SaldoDelegate^ sd,DateTime data)
{
    saldoAttuale=this->CalcolaSaldo(data);
    sd(this);
}
```

Quindi passando al metodo *ElaboraSaldo* l'istanza di delegate *saldoVideo*, verrà indirettamente invocato il metodo *VisualizzaSaldo*, mentre con *saldoFile* il metodo statico *SalvaSaldoSuFile*.

```
conto->ElaboraSaldo(saldoVideo,DateTime::Now);
conto->ElaboraSaldo(saldoFile,DateTime::Now);
```

Come detto, essendo ogni delegate derivato dalla classe *MultiCastDelegate*, è possibile aggiungere più metodi alla catena dei metodi da invocare. La lista di invocazione di un delegate multicast viene creata utilizzando gli overload degli operatori + e += forniti dalla classe *MulticastDelegate*. Ad esempio, una volta creato il delegate saldoVideo, è possibile aggiungere un delegate alla lista d'invocazione ed ottenere un delegate *sd3* con la semplice istruzione:

```
ContoCorrente::SaldoDelegate^ sd3 = saldoVideo + gcnew
        ContoCorrente::SaldoDelegate(ContoCorrente::SalvaSaldoSuFile);
```

oppure ottenere un risultato analogo con

```
saldoVideo += gcnew
        ContoCorrente::SaldoDelegate(ContoCorrente::SalvaSaldoSuFile);
```

In questo modo ad esempio, invocando il metodo ElaboraSaldo e passando come delegate *saldoVideo*, verrà eseguita la catena composta da due metodi, e quindi sarà sia stampato il saldo a video, sia salvato su file. Naturalmente è possibile anche l'operazione inversa, cioè quella di rimozione dalla lista di invocazione, per mezzo degli operatori - e -=.
La lista di invocazione, è ottenibile per mezzo del metodo *GetInvocationList* che restituisce un array di oggetti *MulticastDelegate*, i cui elementi sono disposti nell'ordine in cui verranno invocati.

```
Console::WriteLine("InvocationList di sd3");
for each(MulticastDelegate^ d in sd3->GetInvocationList())
{
    Console::WriteLine("delegate {0}",d->Method);
}
```

Il codice precedente stamperà i nomi dei metodi contenuti nel *MulticastDelegate sd3*, ad esempio:

```
InvocationList di sd3
delegate Void VisualizzaSaldo(ContoCorrente)
delegate Void SalvaSaldoSuFile(ContoCorrente)
```

Un delegate può anche essere invocato direttamente, come se fosse un qualunque metodo. Non è ovvio invece se ci si lascia ingannare dal fatto che un delegate è un oggetto come tanti altri. Come oggetto si può invocare la catena dei metodi "contenuti" in un delegate utilizzando il metodo Invoke, derivato dalla classe MulticastDelegate:

```
sd3->Invoke(conto);
```

Con sintassi invece molto più immediata, e forse inaspettata, basta

utilizzare l'istanza del delegate come se fosse un metodo:

```
sd3(conto);
```

Sia l'Invoke che la chiamata stile metodo, non fanno altro che eseguire tutti i metodi presenti nella catena.

Delegati e interfacce

Forse molti di voi avranno notato una similarità fra i delegate e le interfacce, ed in effetti entrambi permettono la separazione dell'implementazione dalla specifica. Un'interfaccia specifica quali membri una classe deve esporre, membri che poi saranno implementati all'interno della classe stessa in maniera invisibile all'esterno. Un delegate fa la stessa cosa, specificando la firma ed il tipo di ritorno di un metodo, metodo che poi sarà implementato in maniera differente ma compatibile con la firma. Abbiamo già visto come dichiarare le interfacce e come implementarle in una classe, e come utilizzarle all'interno del nostro codice, ad esempio per usufruire della potenza e della flessibilità del polimorfismo. I delegate sono altrettanto utili delle interfacce, ma si prestano molto meglio in situazioni forse più particolari, ad esempio, alcuni casi in cui è ideale utilizzare un delegate sono quelli in cui sappiamo già che un singolo metodo verrà chiamato, o se una classe vuole fornire più metodi per una stessa firma, o se tali metodi verranno implementati come statici, o ancora quando abbiamo a che fare con interfacce utente rispondenti a pattern di programmazione ad eventi.

Eventi

Un evento è una particolare implementazione dei delegate. In parole semplici, gli eventi permetteno ad una classe di scatenare l'esecuzione di metodi che si trovano in altre classi, in attesa appunto di un determinato evento, e senza la necessità che si conosca nulla su tali altre classi e metodi. Tramite il concetto di evento, il framework .NET

nasconde la complessità di basso livello del meccanismo a messaggi. I messaggi sono il meccanismo principale utilizzato dalle applicazioni Windows, ma non solo, per spedire e/o ricevere notifiche di un qualcosa che è avvenuto e che interessa l'applicazione stessa. Ad esempio, quando l'utente interagisce con una finestra cliccando su un pulsante, l'applicazione di cui fa parte la finestra verrà informata di ciò tramite un apposito messaggio. L'idea fondamentale da comprendere è simile al classico modello produttore-consumatore. Un oggetto può generare degli eventi, un altro oggetto viene informato di essi ed intraprende delle azioni. Il mittente dell'evento non sa quale altro oggetto lo riceverà , dunque è necessario un meccanismo che funga da intermediario, che prescinda dalle tipologie di oggetti destinatari dell'evento. Tale meccanismo è proprio quello illustrato nel precedente paragrafo, cioè quello dei delegate.

Generare un evento

Per generare un evento dobbiamo prima definire una classe che contenga le informazioni correlate ad esso. Il framework .NET fornisce una classe base da cui derivare i nostri eventi personalizzati, la classe *System::EventArgs*. Si supponga di avere a che fare con un'applicazione automobilistica, e di voler implementare un meccanismo che effettui il monitoraggio dei giri del motore, avvisando l'utente con un allarme quando il valore è al di sopra di una certa soglia, cioè quando il motore va fuori giri, e viceversa che avverta l'utente che il motore si è spento, perché il regime giri è sceso sotto una certa soglia. Si creeranno dunque due classi diverse, entrambe derivate da *EventArgs*. La prima, *MotoreFuoriGiriEventArgs*, secondo la convenzione che prevede di aggiungere il suffisso *EventArgs* al nome dell'evento, contiene solo due proprietà: *Rpm* è il valore raggiunto dai giri del motore e Message formatta un messaggio di testo da restituire al consumatore dell'evento fuori giri.

```
ref class MotoreFuoriGiriEventArgs: EventArgs
```

```
{
public:
    property int Rpm;

    property String^ Message
    {
        String^ get()
        {
            return String::Format("Il numero dei giri del motore risulta
                                                    essere {0}/min",Rpm);
        }
    }
};
```

Bisogna poi dichiarare un delegate che servirà come gestore dell'evento. Se l'evento non ha dati aggiuntivi è possibile utilizzare il delegate standard *EventHandler*. Per il caso di esempio, se ne creerà uno personalizzato in modo da mostrare la nomenclatura standard utilizzata per i gestori di eventi, e poi si vedrà come utilizzare l'*EventHandler* standard, che non necessita di ulteriori dati per l'evento di *MotoreSpento*:

```
delegate void MotoreFuoriGiriEventHandler(Object^ sender,
                                    MotoreFuoriGiriEventArgs^ e);
```

Il framework .NET utilizza la convenzione di fornire un primo parametro sender, in modo che il gestore possa ricavare anche l'oggetto generatore dell'evento. Il secondo parametro conterrà invece l'evento con gli eventuali dati aggiuntivi che lo caratterizzano.
Per indicare che una classe può scatenare un dato evento si deve aggiungere ad essa un nuovo membro, utilizzando la parola chiave *event*. Insieme ad essa deve essere indicato il delegate che indica la firma che deve utilizzare un metodo gestore di tale evento, per esempio:

```
ref class Motore
{
public:
    event MotoreFuoriGiriEventHandler^ fuoriGiriEvent;
    event EventHandler^ motoreSpentoEvent;
…
}
```

I due membri *fuoriGiriEvent* e *motoreSpentoEvent*, identificano i due eventi che possono essere generati dalla classe, nello stesso modo in cui, come ulteriore esempio, la classe *Button* possiede un campo *Clic*, che indica l'evento di *clic* su di esso.
Il passo successivo è l'aggiunta, sempre alla classe in cui si può verificare l'evento, di un metodo con un nome che deve essere del tipo *OnNomeEvento*. È proprio in tale metodo che viene generato l'evento, invocando il delegate identificato dalla relativa keyword event, e creando un oggetto *EventArgs* contenente eventuali dati aggiuntivi. Nell'esempio dunque sarà:

```
protected:
    //genera l'evento fuori giri
    virtual void OnFuoriGiri(MotoreFuoriGiriEventArgs^ ev)
    {
        FuoriGiriEvent(this,gcnew MotoreFuoriGiriEventArgs(rpm));
    }

    //genera l'evento motore spento
    virtual void OnMotoreSpento()
    {
        MotoreSpentoEvent(this,gcnew EventArgs());
    }
```

Come in questo caso, in genere, il metodo è dichiarato protected e vir-

tual, in tale modo esso potrà essere ridefinito in una classe derivata, ed eventualmente, in questa verrà anche richiamato il metodo della classe base. La ref class *Motore* riportata qui di seguito riprende tutti i concetti esposti finora:

```
ref class Motore
{
private:
	static const int maxRpm=7000;
	static const int minRpm=700;
	bool acceso;

public:
	event MotoreFuoriGiriEventHandler^ FuoriGiriEvent;
	event EventHandler^ MotoreSpentoEvent;
	property int rpm;

	Motore()
	{
		rpm=0;
		acceso=false;
	}

	void Accendi()
	{
		acceso=true;
		this->rpm=800;
	}

	void AumentaRpm()
	{
		if(acceso)
			this->rpm+=100;
```

```
        if(rpm>maxRpm)
            OnFuoriGiri(gcnew MotoreFuoriGiriEventArgs(rpm));
    }

    void DiminuisciRpm()
    {
        if(acceso)
            this->rpm-=100;
        if(rpm<minRpm)
        {
            this->acceso=false;
            OnMotoreSpento();
        }
    }
protected:
    //genera l'evento fuori giri
    virtual void OnFuoriGiri(MotoreFuoriGiriEventArgs^ ev)
    {
        FuoriGiriEvent(this,gcnew MotoreFuoriGiriEventArgs(rpm));
    }

    //genera l'evento motore spento
    virtual void OnMotoreSpento()
    {
        MotoreSpentoEvent(this,gcnew EventArgs());
    }
};
```

La classe *Motore* fornisce due metodi, *AumentaRpm* e *DiminuisciRpm* che nel caso in cui i giri del motore sforino le soglie definite, richiamano i due metodi *OnFuoriGiri* ed *OnMotoreSpento*, e quindi generano gli eventi relativi.

Consumare un evento

Per consumare un evento bisogna prevedere un metodo che venga richiamato al verificarsi dell'evento, e registrare tale metodo come gestore dell'evento. Si implementerà allora una classe *Auto* che contenga un campo di tipo *Motore*, il quale può eventualmente andare fuori giri o spegnersi, e quindi generare gli eventi relativi e che sono stati definiti nel paragrafo precedente. I metodi che si occupano della gestione degli eventi devono avere la stessa firma dei delegate che definiscono gli evento stessi, cioè del delegate *MotoreFuoriGiriEventHandler* e del delegate standard *EventHandler*. Dunque dei buoni gestori dell'evento *FuoriGiriEvent* e dell'evento *MotoreSpentoEvent*, potrebbero essere i due seguenti:

```
void Motore_FuoriGiriEvent(Object^ sender, MotoreFuoriGiriEventArgs^ e)
{
    Console::WriteLine("Evento da {0}:\n{1}",sender->ToString(),e-
                                                            >Message);
    Decelera();
}
void Motore_MotoreSpento(Object^ sender, EventArgs^ e)
{
    Console::WriteLine("Evento da {0}:\nMotore Spento",sender-
                                                            >ToString());
}
```

Essi stampano semplicemente il tipo di evento generato, il messaggio contenuto nell'istanza *MotoreFuoriGiriEventArgs* nel primo caso, ed il tipo dell'oggetto che ha generato l'evento, che in questo esempio sarà sempre un'istanza della classe *Motore*. Ora non ci resta che registrare i due gestori associandoli agli eventi relativi. Per far ciò, C++/CLI, permette di utilizzare gli operatori + e += per aggiungere un gestore all'evento, o anche al contrario - e -= per l'operazione di deregistrazione di un gestore, e ciò è giustificato dal fatto che abbiamo a che fare con delegate e

vale quanto detto quando abbiamo parlato dei delegate multicast. Se la classe Automobile contiene dunque un campo motore di classe *Motore*, sarà sufficiente, ad esempio nel costruttore, scrivere le seguenti istruzioni:

```
Automobile()
{
    motore=gcnew Motore;
    motore->FuoriGiriEvent += gcnew MotoreFuoriGiriEventHandler(this,
        &Automobile::Motore_FuoriGiriEvent);
    motore->MotoreSpentoEvent += gcnew EventHandler(this,
&Automobile::Motore_MotoreSpento);
}
```

Si noti come il delegate venga creato non specificando anche il nome della classe *Motore*, visto che infatti esso era stato definito al di fuori della classe *Motore*. Ciò non costituisce un obbligo, il delegate poteva anche essere dichiarato come membro della classe, ed in genere in qualunque parte di codice in cui è definibile una classe, l'unico vincolo è quello di assicurarne la visibilità. Per completare la nostra classe *Automobile* aggiungiamo tre metodi che mette in moto, aumenta e diminuisca i giri del motore:

```
void MettiInMoto()
{
    if(!motore->acceso)
        motore->Accendi();
}

void Decelera()
{
    motore->DiminuisciRpm();
}
```

```
void Accelera()
{
	motore->AumentaRpm();
}
```

Per verificare come la generazione ed il consumo di eventi funzioni non ci resta che creare una sequenza di chiamate che portino il motore nelle condizioni di fuori giri o di spegnimento:

```
Automobile^ alfagt=gcnew Automobile;
alfagt->MettiInMoto();
alfagt->Decelera();
alfagt->Decelera();
```

Qui il motore sarà al di sotto della soglia minima, e dunque si spegnerà generando l'evento *MotoreSpentoEvent* ed informando ancora la classe *Automobile*, in cui verrà richiamato il gestore *motore_MotoreSpento*.

```
alfagt->MettiInMoto();
for(int i=0;i<50;i++)
	alfagt->Accelera();
```

Qui ad un certo punto il motore supererà la soglia massima generando l'evento *FuoriGiri*, di cui la classe *Automobile* riceverà la notifica mediante la chiamata al gestore *motore_FuoriGiriEvent*.
Il meccanismo di definizione e gestione degli eventi è fondamentale nella creazione delle applicazioni grafiche, quindi è fondamentale comprenderlo e assimilarlo per bene.

GESTIONE DEGLI ERRORI

Non esiste un programma perfetto, nonostante tutta l'attenzione che si

possa mettere nell'evitare errori, possono sempre sfuggire delle situazioni di possibile malfunzionamento, ed altre in cui il programma non si comporta come vorremmo. Non è inoltre sufficiente prevedere tali situazioni e restituire ad esempio un codice di errore, soprattutto in metodi con diversi rami di esecuzione, e diverse cose che possono andare male. Supponiamo ad esempio di voler aprire una connessione di rete, scaricare un file, aprirlo e leggere delle stringhe rappresentanti dei numeri, convertirli in interi, fare delle divisioni, spedire il risultato sulla rete, e mille cose di altro genere. Dovremmo prevedere il fatto che il file non esiste, o che è già in uso, o che non abbiamo il permesso di leggerlo, o che se anche riusciamo a leggerlo esso è corrotto, che la connessione di rete non è disponibile, che la divisione non è eseguibile perchè i numeri sono nulli. Per fortuna, .NET fornisce degli strumenti per gestire situazioni più o meno eccezionali, ed appunto queste funzionalità ricadono in quella che è detta gestione delle eccezioni. Un'eccezione in C++/CLI è un oggetto, derivato in modo diretto o indiretto dalla classe *System::Exception*. Quando si verifica una eccezione, viene creato un tale oggetto contenente, tra le altre, informazioni sull'errore che si è verificato. Uno dei vantaggi, è che l'eccezione non è confinata ad un solo linguaggio, vale a dire che se scriviamo delle librerie in C#, che possono generare delle eccezioni, e utilizziamo tale libreria da codice C++/CLI, è possibile gestire le eccezioni essendo esse oggetti CLI a tutti gli effetti. Eccezioni più specifiche possono inoltre derivare da classi particolari, ad esempio eccezioni che riguardano errori di Input/Output sono in genere derivate dalla classe *IOException*, da cui a loro volta derivano fra le altre *FileNotFoundException* o FileLoadException che riguardano ancora più specificatamente eccezioni nella gestione di file.

Catturare le eccezioni

Una volta che un'eccezione si è verificata, cioè è stata lanciata in un qualche punto del programma, è necessario catturarla e svolgere azioni di recupero. Per far ciò, le parti di codice in cui è prevedibile che una o più eccezioni si possano verificare, vengono racchiuse in blocchi

try/catch/finally. Tali blocchi rappresentano rispettivamente la parte di codice che tenta di eseguire un'azione, quella che eventualmente cattura l'errore, e infine quella che effettua eventuali operazioni di pulizia.
Un simile blocco viene scritto nella seguente maniera:

```
try
{
	//codice che può generare un'eccezione
}
catch(TipoEccezione^ ex)
{
	//gestione dell'eccezione
}
finally
{
	//libera le risorse o svolge altre azioni
}
```

Se all'interno del blocco *try* si verifica un'eccezione del tipo previsto dall'istruzione catch, il controllo del programma passa appunto al blocco catch sottostante, in cui possono essere intraprese le azioni atte a risolvere il problema.
Se nel blocco try non si verifica invece alcun errore, o eventualmente al termine del blocco *catch*, il flusso del programma prosegue nel blocco , in cui in genere vengono liberate delle risorse usate nel *try* precedente, o in genere vengono eseguite delle funzioni previste sia in caso di errore che in caso di esecuzione normale. Ecco un esempio pratico: nel blocco *try* seguente viene tentata la conversione della stringa "abc" in intero, cosa impossibile.

```
try
{
	String^ str="abc";
```

```
    int i=System::Convert::ToInt32(str);
}
catch(Exception^ ex)
{
    Console::WriteLine("Si è verificato un problema:"+ex->Message);
}
```

Al momento dell'istruzione *Convert::ToInt32* il flusso di esecuzione del programma verrà interrotto e passerà dentro il blocco *catch*, che cattura in questo caso ogni tipo di eccezione, dato che ogni eccezione deriva dalla classe *Exception*. In questo caso particolare si verificherà una eccezione di classe *FormatException*, ed il messaggio stampato sarà che la stringa di input non ha un formato corretto. I blocchi catch possono essere anche più di uno in cascata, in maniera da gestire eccezioni di tipo differente, ad esempio supponiamo che nel blocco *try* tentiamo di aprire un file, può accadere che il file non esista, oppure che il file sia vuoto e si tenti di leggere una riga di testo:

```
void TestTryCatchFinally()
{
    StreamReader^ sr=nullptr;
    try
    {
        sr=File::OpenText("C:\\temp\\filevuoto.txt");
        String^ str=sr->ReadLine()->ToLower();
        Console::WriteLine(str);
    }
    catch(FileNotFoundException^ fnfEx)
    {
        Console::WriteLine(fnfEx->Message);
    }
    catch(NullReferenceException^ flEx)
    {
```

```
            Console::WriteLine(flEx->Message);
      }
      finally
      {
            if(sr!=nullptr)
                  sr->Close();
      }
}
```

Se il file non viene trovato l'esecuzione salterà al primo blocco catch, mentre nel caso in cui il file sia vuoto, al tentativo di leggere una riga con il metodo *sr->ReadLine()*, verrà restituito il valore nullptr, e dunque nel convertire tale stringa nulla in minuscolo verrà generata un eccezione *NullReferenceException*, gestita dal secondo blocco catch. Alla fine, comunque vada, si passa dal blocco finally, che eventualmente chiude l'oggetto *StreamReader*. Nel gestire più tipi di eccezioni, con più blocchi *catch*, è necessario prestare attenzione all'ordine dei tipi di eccezione. Infatti è obbligatorio gestire per primi le eccezioni più specifiche, cioè che fanno parte di una catena di derivazione da *System::Exception* più lunga. Se non fosse così infatti l'eccezione potrebbe essere catturata da un catch precedente. Il compilatore comunque impedisce una simile eventualità, ad esempio:

```
try
{...}
catch(Exception^ ex)
{
      Console::WriteLine(ex->Message);
}
catch(FileNotFoundException^ fnfEx)
{
      Console.WriteLine(fnfEx->Message);
}
```

Il messaggio che il compilatore sarà in questo caso del tipo:

```
Error C4286: System::IO::FileNotFoundException ^: è catturata da una
                                          classe base (System.Exception)
```

Nella gestione delle eccezioni è possibile omettere sia il blocco *catch* che il blocco *finally*, ma non entrambi contemporaneamente. In particolare la situazione in cui è presente il solo blocco *finally* viene utilizzata per garantire che vengano eseguite certe istruzioni prima di uscire da un metodo, ad esempio il blocco *try* potrebbe contenere più istruzioni return, e quindi più punti di uscita. Quando si incontra il primo dei return il programma prima di terminare il metodo passa dal blocco *finally*:

```
public int TestTryFinally()
{
    int i=2;
    try
    {
        switch(i)
        {
            case 2:
                return 4;
            case 3:
                return 9;
            default:
                return i*i;
        }
    }
    finally
    {
        Console::WriteLine("Prima di uscire passa dal finally");
    }
}
```

Lanciare le eccezioni

Essendo una classe come le altre, da *Exception* è possibile derivare le proprie eccezioni, e se è possibile fare ciò, è naturalmente possibile lanciare le eccezioni personalizzare dalle proprie applicazioni. È bene lanciare un'eccezione solo quando il programma si trova in una situazione inaspettata o imprevista, in quanto l'elaborazione di tali istruzioni è più pesante di un normale flusso di esecuzione.
In parole povere, se in un dato punto del codice viene eseguita una divisione, e se può capitare che il valore per cui dividere sia 0, è meglio controllarlo, piuttosto che eseguire la divisione e lanciare un'eccezione.
Il meccanismo per cui un'eccezione viene generata, viene chiamato *throwing*.
Quando si verifica l'evento eccezionale, viene eseguita un'istruzione del tipo:

```
throw gcnew TipoEccezione(parametri);
```

L'istruzione *throw* ha la funzione di generare e lanciare un'eccezione del tipo specificato, e se tale *throw* viene eseguito da una parte di codice eseguita all'interno di un blocco *try*, entra in funzione la cattura dell'eccezione, sempre che il tipo di eccezione sia stato previsto in un blocco *catch*.

```
void TestThrow()

{
    try
    {
        throw gcnew Exception("Genero un'eccezione");
    }
    catch(Exception^ ex)
    {
        Console::WriteLine(ex->Message);
```

```
    }
}
```

in questo caso l'eccezione verrà catturata e verrà stampato il messaggio "Genero un'eccezione" contenuto nell'istanza ex della classe *System::Exception*. È possibile utilizzare l'istuzione throw anche al di fuori di un blocco *try*, cioè in una parte qualunque di codice, ad esempio in un metodo. In questa maniera, al verificarsi di un'eccezione, non trovando il blocco catch per gestirla, il CLR risalirà lo stack delle chiamate fino a trovarne uno, o al limite fino a quando verrà restituito un messaggio di eccezione non gestita. Ad esempio si supponga di scrivere una liberia matematica con un metodo che effettua la divisione fra due numeri:

```
int Dividi(int a,int b)
{
    if(b==0)
        throw gcnew DivideByZeroException("Eccezione generata da
                                                        Dividi(a,b)");
    return a/b;
}
```

nel metodo l'eccezione può essere generata, ma non è gestita, allora utilizzando questo metodo da qualche altra parte nel codice oppure in una applicazione che usa la libreria, si dovrà gestire l'eventuale eccezione:

```
public void ChiamaDividi()
{
    try
    {
        Dividi(5,0);
    }
    catch(DivideByZeroException^ ex)
```

```
    {
        Console::WriteLine(ex->Message);
    }
}
```

L'eccezione generata nel metodo *Dividi* viene propagata e gestita dal metodo chiamante, cioè il metodo *ChiamaDividi*. Come detto prima, il *throw* può essere eseguito anche all'interno di un blocco *catch*, in questo caso si parla di eccezione rilanciata, ed esattamente viene rilanciata al metodo che ha chiamato quello in cui se è verificata, e così via risalendo lungo lo stack delle chiamate.

```
void ChiamaDividi()
{
    try
    {
        Dividi(5,0);
    }
    catch(DivideByZeroException^ ex)
    {
        Console::WriteLine("Rilancio l'eccezione "+ex->GetType());
        throw ex;
    }
}
```

Dopo aver stampato un messaggio con informazioni sull'eccezione l'eccezione verrà rilanciata.

Try innestati

Oltre a poter propagare l'eccezione ad un altro metodo, come negli esempi precedenti, è possibile scrivere dei blocchi annidati di try, in questa maniera un'eccezione generata all'interno di un blocco *try*, nel caso in cui il corrispondente *catch* non la gestisca, sarà propagata al bloc-

co *try* esterno, ed eventualmente gestita dai blocchi catch di questo e così via.

```
void NestedTry()
{
	array<int> ^arr=gcnew array<int>{4,2,0};
	int dividendo=100;
	for(int i=0;i<4;i++)
	{
		try
		{
			try
			{
Console::WriteLine("{0}/{1}={2}",dividendo,arr[i],dividendo/arr[i] );
			}
			catch(DivideByZeroException^ de)
			{
				Console::WriteLine(de->ToString());
				Console::WriteLine(de->Message);
			}
		}
		catch(IndexOutOfRangeException^ ie)
		{
			Console::WriteLine(ie->Message);
		}
	}
}
```

Nel codice precedente abbiamo due *try* innestati, in quello più interno viene eseguita una divisione fra interi (dividendo / arr[i]), l'eventuale eccezione *DivideByZeroException* viene gestita dal corrispondente catch. Ma gli operandi della divisione, in particolare il divisore viene preso da un array di tre interi, ciclando però con un for il cui indice varia da 0

a 3. All'ultimo accesso all'array, l'indice supererà i limiti dell'array, generando dunque una eccezione *IndexOutOfRangeException*, la quale non essendo gestita nel blocco try interno, verrà propagata a quello più esterno, in cui invece esiste un catch apposito. Provando a compilare il codice ed eseguendo, si avranno dunque le prime due divisioni eseguite correttamente, e due eccezioni:

```
100/4=25
100/2=50
DiveideByZeroException
Index was outside the bounds of the array.
```

La classe *System.Exception*

System::Exception costituisce la classe base per ogni altro tipo di eccezione in C++/CLI, e .NET in generale. La classe è in se stessa molto semplice, essendo costituita da una serie di proprietà pubbliche, utilizzabili per ricavare vari tipi di informazioni sull'eccezione stessa. Tali proprietà sono riassunte nella tabella seguente:

Proprietà	Descrizione
String^ Message	Restituisce una stringa che descrive l'eccezione.
String^ Source	Restituisce o imposta il nome dell'applicazione o dell'oggetto che ha generato l'eccezione.
String^ StackTrace	Restituisce la rappresentazione dello stack di chiamate al momento dell'eccezione.
String^ HelpLink	Restituisce o imposta il link alla documentazione sull'eccezione generata.
Exception^ InnerException	Restituisce l'istanza di System.Exception che ha causato l'eccezione corrente, cioè se un'eccezione A è stata lanciata da una precedente eccezione B, allora la proprietà InnerException di A restituirà l'istanza B.
MethodBase^ TargetSite	Restituisce il metodo in cui è stata generata l'eccezione.

Il seguente metodo mostra come utilizzare queste proprietà:

```
void PrintExceptionInfo(Exception^ ex)
{
    Console::WriteLine("Message: {0}",ex->Message);
    Console::WriteLine("Source: {0}",ex->Source);
    Console::WriteLine("StackTrace: {0}",ex->StackTrace);
    Console::WriteLine("HelpLink: {0}",ex->HelpLink);
    Console::WriteLine("InnerException: {0}",ex->InnerException);
    Console::WriteLine("Method Name: {0}",ex->TargetSite->Name);
}
```

Eccezioni personalizzate

Ogni sviluppatore può creare le proprie classi di eccezioni per rappresentare meglio una situazione che può generarsi nella propria applicazione e quindi per rispondere meglio alle esigenze di gestione della stessa. Avendo già visto come derivare una classe da una esistente, si dovrebbe essere in grado di implementare le proprie eccezioni, derivando una classe dalla *System::Exception*, o da una eccezione il più vicino possibile a quella che si vuol implementare, ad esempio se si tratta di un'eccezione che riguarda l'input/output sarebbe una buona scelta derivarla dalla *IOException*.

```
ref class EccezionePersonalizzata: Exception
{
}
```

già questo potrebbe bastare per lanciare un'*EccezionePersonalizzata*, in quanto il verrà fornito automaticamente un costruttore di default senza parametri.

```
try
{
```

```
    throw gcnew EccezionePersonalizzata();
}
catch(EccezionePersonalizzata^ ep)
{
    Console::WriteLine(ep->Message);
}
```

Il codice precedente stamperà un output del tipo:

```
Generata eccezione di tipo 'EccezionePersonalizzata'.
```

Naturalmente è possibile fornire maggiori dettagli alle proprie eccezioni, partendo dall'implementare diversi costruttori, come nella seguente classe *MiaException*:

```
ref class MiaException:InvalidOperationException
{
public:
    MiaException():InvalidOperationException() {
    }
    MiaException(String^ msg):InvalidOperationException(msg) {
    }
    MiaException(String^ msg,Exception^
                    inner):InvalidOperationException(msg,inner) {
    }
};
```

Testando questa nuova eccezione nel codice seguente:

```
try
{
    try
    {
```

```
            String^ str="";
            int a=1/str->Length;
        }
        catch(DivideByZeroException^ ex)
        {
            throw gcnew MiaException("Operazione impossibile",ex);
        }
}
catch(MiaException^ me)
{
        Console::WriteLine("Message {0}",me->Message);
        Console::WriteLine("InnerException: {0}",me->InnerException-
                                                                >Message);
}
```

Si otterrebbe ad esempio questo messaggio:

```
Message Operazione impossibile
InnerException: Tentativo di divisione per zero.
```

Nulla vieta, sviluppando una libreria propria di classi di eccezioni personalizzate, di aggiungere campi e metodi ad hoc, ad esempio prevedere un codice numerico d'errore ed un metodo per ricavarlo da un'istanza dell'eccezione stessa, o ancora meglio dei metodi per recuperare dalla situazione d'errore nel modo più appropriato, in quanto, avendo progettato lo sviluppatore stesso l'eccezione, nessuno può sapere meglio di lui come trattare i casi in cui si verifica.

LE CLASSI DEL FRAMEWORK

Il framework .NET è composto da centinaia di classi, organizzate in più di 100 namespace. Ogni classe deriva dalla super classe *System::Object*, e come consiglio non si può che affermare che la conoscenza del framework

è fondamentale per programmare in .NET. Se si ha bisogno di scrivere o leggere un file, di accedere al registro, di creare un pulsante, o altre normali esigenze, con alta probabilità si troverà fra le librerie del framework la classe adatta.
Per referenziare un assembly, e poterne utilizzare i tipi e i metodi, si utilizza la direttiva #using. Per esempio, se si vuole utilizzare la classe Form del namespace *System::Windows::Forms*, contenuta nell'assembly *System.Windows.Forms.dll*, si dovrà scrivere all'inizio del file la direttiva:

```
#using < System.Windows.Forms.dll>
```

Poi, come già visto, si aggiungerà l'istruzine using, da non confondere dalla direttiva precedente:

```
using namespace System::Windows::Forms;
```

Se si utilizza un IDE, come Visual Studio, non è però necessario preoccuparsi dei vari riferimenti agli assembly.

La classe System::Object

Tutto in .NET è un oggetto, ed ogni classe deriva dalla classe *System::Object*. Se dunque si scrive

```
ref class Pippo
{
	…
}
```

è come scrivere

```
ref class Pippo: System::Object
{
	…
```

```
}
```

La classse *System::Object* fornisce dei metodi public o protected, statici e di istanza, che dunque ogni altra classe ref possiede e che può eventualmente ridefinirne con un override se essi sono definiti anche come virtual. La tabella seguente elenca tali metodi:

Metodo	Descrizione
virtual string ToString()	Restituisce una stringa che rappresenta l'oggetto.
virtual int GetHashCode()	Restituisce un intero, utilizzabile come valore di hash, ad per la ricerca dell'oggetto in un elenco di oggetti.
virtual bool Equals(Object^ o)	Effettua un test di uguaglianza con un'altra istanza della classe.
static bool Equals(Object^ a, Object^ b)	Effettua un test di uguaglianza fra due istanze della classe.
static bool ReferenceEquals(Object^ a, Object^ b)	Effettua un test di uguaglianza per verificare se due riferimenti si riferiscono alla stessa istanza della classe.
Type^ GetType()	Restituisce un oggetto derivato da System.Type che rappresenta il tipo dell'istanza.
Object^ MemberwiseClone()	Effettua una copia dei dati contenuti nell'oggetto, creando un'altra istanza.
virtual void Finalize()	Distruttore dell'istanza.

Il metodo ToString

Il metodo *ToString()* è probabilmente uno dei più utilizzati in qualsiasi tipo di applicazione, in quanto serve a fornire una rappresentazione testuale del contenuto di un oggetto. Esso è un metodo *virtual* nella classe *System::Object*, dunque ogni classe può fornire un override di esso, in modo da restituire una stringa significativa. Ad esempio i tipi nume-

rici predefiniti di C++/CLI forniscono tale override in modo da restituire il valore sotto forma di stringa.

```
int i=100;
String^ str=i.ToString(); // restituisce "100"
```

se non ridefiniamo il metodo nelle nostre classi, verrà invocato il metodo della classe *System::Object*, che restituirà una rappresentazione più generica.

```
ref class Studente
{
private:
    int matricola;
    String^ cognome;
    String^ nome;
public:
    Studente(int m, String^ n, String^ c)
    {
        matricola=m;
        cognome=c;
        nome=n;
    }
};

Studente^ studente=gcnew Studente(1,"Antonio","Pelleriti");
Console::WriteLine(studente);
```

La chiamata a *Console::WriteLine(studente)* invoca al suo interno il metodo *studente->ToString()*, e in questo caso stamperà solo il nome della classe stessa, mentre magari ci si aspetterebbe una stringa che contenga matricola, nome e cognome dello studente. Per far ciò, è necessario un override del metodo *ToString()*:

```
virtual String^ ToString() override
{
    return String::Format("matr. {0} - {1} {2}",matricola, cognome,nome);
}
```

I metodi Equals e ReferenceEquals

I metodi *Equals* effettuano il confronto di due istanze, e sono fondamentali in quanto vengono richiamati in diverse altre classi per testare l'uguaglianza di due oggetti, ad esempio nelle collezioni standard del .NET framework, il metodo di istanza bool *Equals(Object^ o)* viene utilizzato per verificare se esse contengono o meno una certa istanza. La classe *System::Object* tuttavia fornisce un'implementazione del metodo Equals che verifica semplicemente l'uguaglianza dei riferimenti. Senza fornire un override del metodo, vediamo come si comporterebbe il seguente codice:

```
Studente^ st1=gcnew Studente(1234,"pinco", "pallino");
Studente^ st2=st1;
Studente^ st3=gcnew Studente(1234,"pinco", "pallino");
Console::WriteLine(st3->Equals(st1)); //stampa false
```

L'ultima riga stamperà il risultato false, dunque *st3* e *st1*, pur rappresentando logicamente lo stesso *Studente*, vengono considerati diversi, in quanto si riferiscono a due istanze in memoria distinte della classe *Studente*. Se invece scriviamo un metodo ad hoc nella classe *Studente*, facendo l'override del metodo *Equals* che ad esempio confronti la matricola, andrà scritto così:

```
virtual bool Equals(Object^ obj) override
{
    if(dynamic_cast<Studente^>(obj)!=nullptr)
    {
        return this->matricola==(dynamic_cast<Studente^>(obj))->
```

```
                                                    matricola;
    }
    return false;
}
```

Otterremmo che il risultato del test di uguaglianza *st3->Equals(st1)* restituirebbe *true*. Potrebbe essere necessario comunque dovere confrontare che siano uguali due riferimenti, cioè che in realtà due variabili referenzino la stessa istanza in memoria, a tal scopo la classe *System::Object* fornisce il metodo statico *ReferenceEquals*, che verifica appunto l'uguaglianza dei riferimenti di due oggetti. Ad esempio:

```
Studente^ st4=gcnew Studente(1234,"pinco", "pallino");
Studente^ st5=st4;
Console::WriteLine("Object::ReferenceEquals(st4, st5)=
                        "+Object::ReferenceEquals(st4,st5));//true
st5=gcnew Studente(1234,"pinco", "pallino");
Console::WriteLine("Object::ReferenceEquals(st4,st5)=
                        "+Object::ReferenceEquals(st4,st5));//false
```

La prima chiamata a *Object::ReferenceEquals* restituisce *true* in quanto *st4* e *st5* sono due handle dello stesso oggetto *Studente* in memoria heap, mentre dopo avere costruito una nuova istanza ed assegnatala a *st5*, la stessa chiamata restituisce false, in quanto ora abbiamo appunto due istanze diverse della classe *Studente*.

Il metodo GetHashCode

Il metodo *GetHashCode* restituisce un valore intero, utilizzabile per lavorare con collezioni tipo tabelle hash, e memorizzare oggetti tipo chiave/valore. Una hash table è formata da tante locazioni, e la locazione in cui memorizzare o ricercare una coppia chiave/valore è determinata dal codice hash ricavato dalla chiave. Ad esempio quando si deve ricercare un dato oggetto, viene ricavato il codice hash dalla chiave, e nella po-

sizione indicata da tale codice, verrà poi ricercata la chiave stessa, e quindi se trovata tale chiave, può essere ricavato il valore corrispondente. In particolare, la *Base Class Library* fornisce una classe *Hashtable*, che utilizza il metodo *GetHashCode* per ricavare il codice *hash*, ed il metodo *Equals* per confrontare gli oggetti da memorizzare o memorizzati. Ed infatti se effettuiamo nella nostra classe un override del metodo *Equals*, il compilatore ci avviserà con un warning se non si implementerà anche l'override del metodo *GetHashCode*.
L'implementazione del metodo richiede il ritorno di un valore int, e dunque è necessario trovare un algoritmo che restituisca dei valori hash con una distribuzione casuale, con una certa velocità per questioni di performance, e naturalmente che restituisca valori hash uguali per oggetti equivalenti. Ad esempio, due stringhe uguali dovrebbero restituire lo stesso codice hash. La documentazione .NET contiene un esempio abbastanza significativo per una struct *Point* così fatta:

```
using namespace System;
public value struct Point
{
public:
  int x;
  int y;

  virtual int GetHashCode() override
  {
    return x ^ y;
  }
};
```

Il metodo precedente restituisce lo *XOR* fra le coordinate *x* e *y* del punto. Qualche sviluppatore preferisce richiamare direttamente il metodo della classe base *Object*, soprattutto se le funzionalità di hash non sono necessarie:

```
public override int GetHashCode()
{
   return base.GetHashCode();
}
```

Il metodo GetType

Il metodo *GetType()* restituisce un'istanza della classe *System::Type*, che può essere utilizzata per ottenere una grande varietà di informazioni a run-time sul tipo corrente di un oggetto, ad esempio il namespace, il nome completo, i metodi di una classe, quindi è il punto di accesso alla cosiddetta tecnologia di *Reflection*, la quale consente di esplorare il contenuto di un tipo qualunque, ma questa è un'altra storia.

```
System::Windows::Forms::Button^ button=gcnew
                              System::Windows::Forms::Button;
Type^ t=button->GetType();
Console::WriteLine("Il tipo di button è "+t->FullName);
```

naturalmente ciò vale anche per i tipi valore:

```
i=0;
Console::WriteLine("Il tipo di i è "+i->GetType()->FullName);
```

CONCLUSIONI

Durante il susseguirsi delle pagine e degli argomenti del libro appena concluso, ho cercato di affrontare ogni parola in maniera da permettere a tutti i lettori, e soprattutto a chi è alle prime armi, di impadronirsi nel modo meno traumatico possibile delle caratteristiche del linguaggio. Naturalmente, nessun linguaggio si può sviscerare in così poche pagine, e quindi spero solo di aver dato un minimo delle nozioni necessarie a poter affrontare lo studio su un testo più completo, o direttamente lo sviluppo di un'applicazione. Per chi avesse domande, critiche o suggerimenti, sono comunque disponibile via email (antonio.pelleriti@dotnetarchitects.it) o sul mio sito internet *www.dotnetarchitects.it*.

Note biografiche

Antonio Pelleriti, è ingegnere informatico, sviluppa software da più di dieci anni e si occupa di .NET sin dalla prima versione Beta. È chief software architect di DynamiCode s.r.l., Software Factory in cui progetta e sviluppa soluzioni custom ed in outsourcing (*www.dynamicode.it*). La passione per i linguaggi di programmazione lo ha portato, per il momento, fino a C++/CLI, che è l'ultimo dei linguaggi, in termini di tempo, utilizzati nello sviluppo di applicazioni Windows.

APPLICAZIONI .NET CON C++/CLI

Autore: Antonio Pelleriti

EDITORE
Edizioni Master S.p.A.
Sede di Milano:Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

Art Director: Paolo Cristiano
Responsabile grafico di progetto: Leonardo Cociero
Responsabile area tecnica: Giancarlo Sicilia
Illustrazioni: Tonino Intieri
Impaginazione elettronica: Francesco Cospite

Servizio Clienti
☎ **Tel. 02 833836 - Fax 02 83383610**
@ **e-mail: customercare@edmaster.it**

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Gennaio 2009